SWDIV-HDBK-7 (Rev. 1)
1 November 2002

SOFTWARE METRICS PROGRAM HANDBOOK

DEPARTMENT OF THE NAVY
COMMANDER
NAVAL AIR SYSTEMS COMAND
47123 Buse Road
Patuxent River, MD 20620

FOREWORD

This handbook was created to provide an introduction to the collection and use of software management measurements & metrics. It is intended as a supplement and to provide detailed guidance for the requirements of reference (a) NAVAIR INSTRUCTION 5234.5 NAVAL AIR SYSTEMS COMMAND METRICS FOR SOFTWARE INTENSIVE PROGRAMS. It is also intended to be compatible and compliant with reference (b) IEEE/EIA 12207 The Software Life Cycle Process that has replaced MIL-STD-498 and DOD-STD-2167 as the DOD standard for the Software Life Cycle Development Process. Specific guidance in this document is also intended to be compliant with and supplement reference (c) Practical Software Measurement (PSM). PSM is an implementation of both the CMMI (Capability Maturity Model Integrated) Measurement requirements and with reference (d) ISO/IEC 15939, Software Engineering - Software Measurement Process. Reference (c) is strongly recommended to those interested in gaining a more detailed and comprehensive understanding of all aspects of implementing and establishing a Software Measurement Program which is more than can be covered within the scope of this document. Used together with other software development best practices, an effective software metrics program provides essential insight into the program's status in achieving its performance, cost and schedule requirements.

> **"***When you can measure what you are speaking about and express it in numbers, you know something about it; but when you can not express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science***."**
>
> Lord Kelvin (3 May 1883)

For copies or comments the following media are available:

By mail, contact:	NAVAIR (41F000D) Software Resource Center Knox Rd, Build 1494 China Lake, CA 93555-6100
By telephone (voice), use:	Software Resource Center, (760)939-7086, DSN 437-7086
By FAX (TELEX), use:	Software Resource Center, (760)939-0150, DSN 437-0150
By Email:	NW41P3R@navair.navy.mil
To contact the author use:	Rick Holcomb holcombra@navair.navy.mil (301)342-2450, DSN 342-2450

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDIXES

THIS PAGE INTENTIONALLY LEFT BLANK

1. OVERVIEW - METRICS PROGRAM

1.1 INTRODUCTION

According to the 2000 study by the Standish Group:

- *23% of software development projects fail.*
- *49% were challenged. In the challenged group schedules over ran on average by 63%, cost over ran by 45%, and only 67% of the originally desired functionality was provided.*
- *Only 28% were successful.*

Other studies show similar or worse results. Imagine what America would look like if the construction industry had a similar success rate. One can assume that all of these programs when initiated expected to be successful. What happened? In most cases the failure to implement an effective metrics program was at least part of the reason. Attempting to execute a software development, or any other engineering development effort, without empirical, quantitative data upon which to judge the progress and status of the program is akin to attempting to fly an aircraft without any instruments or navigation systems. It may be acceptable for a small software development or ultra-light aircraft, but it is not a good idea for a large software development or trying to fly a 747 across the Pacific Ocean.

A measurement program is the essential first step to any effort to implement software process improvement efforts seeking to achieve greater performance at a lower cost and on a shorter schedule. Without an effective metrics program, how can it be determined if these process improvements are actually having a desirable effect on the program? The answer is that it can't; in fact it may be having just the opposite effect.

Many program managers use the high cost of a metrics program, often cited as 2 - 5%[1] of program costs as a reason for not implementing such a program. However, without a metrics program, these managers have no means to either develop a realistic cost, schedule, or performance goals for the program, or to identify deviations from their plans early enough to take meaningful corrective action. Thus, without a metrics program, these program managers really have no idea how much their program will cost or what is the current status of the program is. Its not a matter of being able to afford a metrics program; a large software development can't afford not to use metrics.

1.2 REFERENCES

Top level requirements for use of metrics can be found in the following documents. In the event of a conflict between this handbook and formal guidance, the formal guidance will always take precedence.

(a) NAVAIR INSTRUCTION 5234.5 NAVAL AIR SYSTEMS COMMAND METRICS FOR SOFTWARE INTENSIVE PROGRAMS.

(b) IEEE/EIA 12207 Software Life Cycle Process

[1] Practical Software & Systems Measurement Objective Information for Decision Makers, PSM Overview Version 5.0.

(c) Practical Software Measurement, Objective Information for Decision Makers, John McGarry, David Card, Cheryl Jones, Beth Layman, Elizabeth Clark, Joseph Dean, Fred Hall, Addison Wesley 2002.

(d) ISO/IEC 15939, Software Engineering - Software Measurement Process

(e) Software Size Measurement: A Framework for Counting Source Statements CMU/SEI-92-TR-20 ESC-TR-92-020

(f) Practical Software Measurement, a Foundation for Objective Program Management, version 3.1a, Department of Defense Implementation Guide, PSM Addendum, 17 April 1998. Available at www.psmsc.com.

(g) Capability Maturity Model® Integration (CMMI^{SM}) for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1) CMU/SEI-2002-TR-012 ESC-TR-2002-012. Measurement and Analysis Process Area

1.3 TERMINOLOGY

This handbook makes a subtle differentiation between software measures and software metrics:

- *Measurements are the (raw) data elements. They are directly observable quantities that can be counted, such as lines of code, labor hours, and labor months. Under PSM, reference (c), these are referred to as "Base Measures".*
- *As used in this manual, a Metric is considered some measure or combination of measures that reflect some aspect of an item in which one has an interest. For example, the growth of the size of a software module over time might be one metric of interest to the manager of a software development effort. Metric as it is used here is the same as a Derived Measure and/or an Indicator as referred to in PSM, reference (c).*

Terminology in this handbook is consistent with Reference (a), (b) and (c) as much as possible. Appendix C provides a cross reference of terminology used by Reference (b) IEEE/EIA 12207 to older DoD Software Life Cycle Processes such as MIL-STD-498 and DOD STD-2167A.

1.4 INFORMATION NEEDS

Metrics and Measurements must be selected based on the information needs of the program. There are essentially three types of Information Needs:

- ✓ *Program Objectives and Commitments – Operational Requirements Document (ORD) Key Performance Parameters (KPPs), critical specification requirements, fiscal and schedule constraints, product acceptance criteria, external agreements and interface requirements, etc..*
- ✓ *Process Improvement Goals – measure effectiveness of process improvement efforts.*
- ✓ *Problems – areas of concern that a project is currently experiencing or is relatively certain to experience.*

- ✓ *Risks – areas of concern that could occur, but are not certain to occur.*
- ✓ *Lack of Information – areas of concern where the available information is inadequate to reliably predict the project impact.*

The software metrics program must collect and analyze measurements and metrics that will provide visibility into program Information Needs. Metrics which do not provide such insight are useless and a waste of effort to collect. The PSM methodology is based on this concept.

At a minimum, all programs should assume they will experience problems in the areas of performance, cost and schedule. The measurements set found in this handbook provide examples and suggestions of the minimum core measurements and metrics needed for visibility into and mitigation of any problems arising in the areas performance, cost and schedule.

- ✓ *Additional measurements and metrics, not discussed in this handbook, may be required to provide visibility into other program information needs.*
- ✓ *As the program information needs change over the development life cycle, so to must the measurements and metrics change in order to continue to provide insight into current and pertinent issues for the program.*
- ✓ *The buyer (Government Program Office and/or Software Support Activities (SSAs)) and the developer (Industry, SSAs, or other source) must work closely to identify the current information needs of the program and ensure that only measurements and metrics are collected which provide insight into those information needs.*

1.5 METRICS INTERRELATIONSHIPS; A COMMON FRAMEWORK

For the purposes of this handbook, metrics are grouped into seven categories as follows: (1) Requirements, (2) Size, (3) Staffing, (4) Capacity, (5) Quality, (6) Schedule and (7) Cost.

Figure 1-1 depicts the seven categories and the associated software metrics in each category.

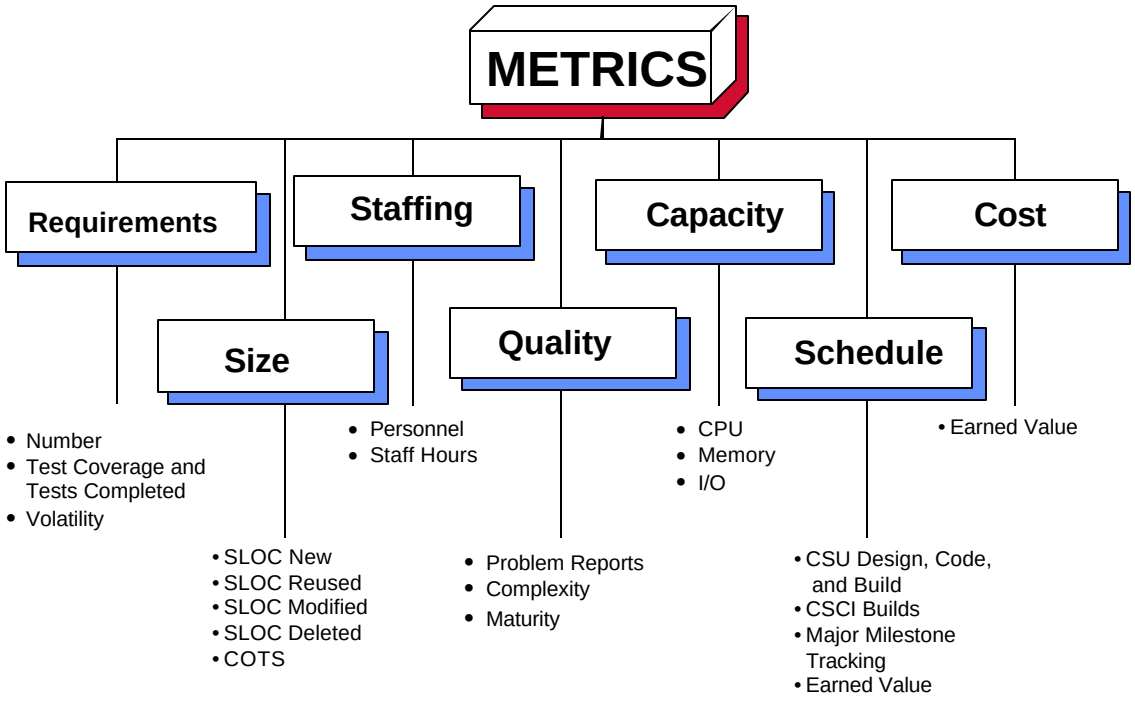


FIGURE 1-1. Software Metrics.

Figure 1-2 shows the relative relationship between the categories and associated metrics. Such interrelationships exist between most metrics.

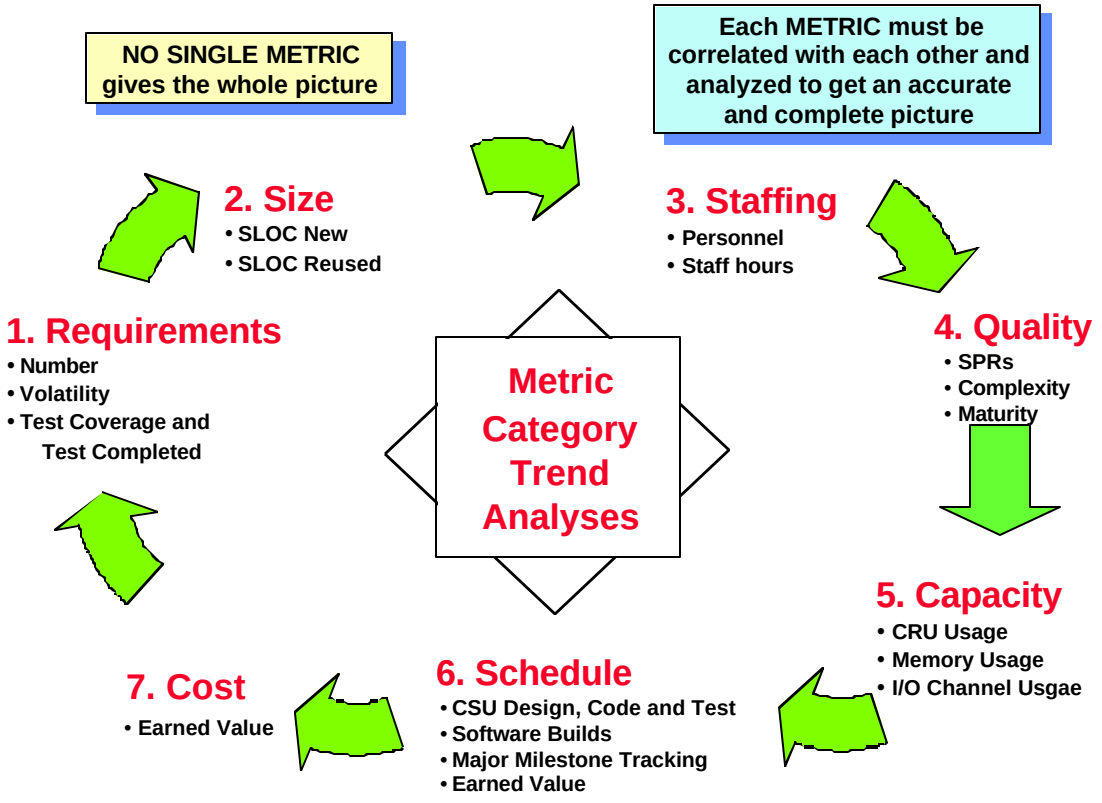


FIGURE 1-2. Metrics Categories and Relationships.

1.6 USE OF METRICS & MEASUREMENTS

References (c), (d) and (g) provides guidance for integrating the use of measurement and analysis into other project activities. Integrating measurement and analysis into other activities can increase both the accuracy and benefits of the selected measures.

Metrics & measurements can, and should, be tailored and collected in a way that minimizes costs and intrusions into the developer's normal software engineering practices. Where the developer has an established set of measurements, that set should be reviewed for suitability and should be used when they meet the requirements of providing visibility into the software buyer's information needs. Modifications and additions to the developers established measurement and metrics set should only be required if the costs of these changes are justified by the buyer's information needs. However, needed measurements should always be identified as a contract deliverable.

Metrics & measurement requirements placed on the developer should also be extended to the developer's subcontractors performing software development. Consideration should be given to standardizing measurement definitions and collection methods across all software development subcontractors.

2. METRICS

2.1 INTRODUCTION

The following metrics are provided as examples and guidance in developing and implementing a metrics program. Each of these metrics will provide visibility into different aspects of the software development process. The specific metrics in this section have been selected for discussion due to their ability to provide insight and visibility into the three primary problem areas of any software development project:

1. Quality
2. Cost
3. Schedule

Note:	*The following measurements and metrics are intended as examples and guidance and not as firm requirements on exactly what measurements & metrics must be collected and analyzed. If the developer uses alternative measurements & metrics which meet the buyer's information needs, than the developer's measurements & metrics should be utilized in order to avoid the cost of perturbing the software development environment. The developer should be forced to collect new measurements and metrics only when the benefit of doing so outweighs the cost of not meeting the information need.*

2.2 REQUIREMENTS

2.2.1 Purpose

Requirements are the primary driver of the size, schedule, effort, and ultimately cost of a software development effort. Requirements can change for a program in three ways:

1. New or added requirements. These will always increase cost and schedule. The later they are added into a program, the greater the cost and schedule increase will be due to rework required in previously completed requirements documents, design documents, code and testing.
2. Deleted requirements. These may result in decreases or increases in cost and schedule depending on when they are deleted. The later they are deleted, the less the savings. At very late points in the development, deletion will actually increase cost due to the rework required to remove these requirements from requirements documents, design documents, code and retesting.
3. Modified or changed requirements. These may increase or decrease the cost depending on the change. The later in the development the change is made, the more likely that it will cause a cost increase. Changes in requirements during later development phases will virtually always increase cost and schedule due to rework of documents, code and retest. Even a change resulting in a decrease in performance can result in cost and schedule increases if it is made late in the development cycle.

The sources of software requirements changes are also critical and can be divided into two categories:

1. User generated changes in requirements. Such changes will generally require an Engineering Change Proposal (ECP) and a subsequent modification to the contract with the developing agency. Such changes are virtually guaranteed to cause an increase in cost. Deletions, made early in the development may result in cost reductions, but keep in mind that cost incurred prior to the deletion of a requirement are sunk and will not be recovered by the deletion. Proposals for "no cost" changes to software requirements should be treated with extreme caution, especially if they are adding capability. Unless savings are being achieved via deletion of other functionality, there is no such thing as a "no cost" change. It is essential that users and resource sponsors understand the impact on program, cost, schedule and other functional requirements of demands for the implementation of additional functionality.
2. Changes in Software Requirements Description (SRD). The SRD expands higher-level systems requirements into detailed software requirements from which design, code and test can be performed. The SRDs will be developed during the requirements analysis phase. Any changes to user requirements will also change the SRDs. The SRD requirements will also change in later development phases if the requirements were incorrectly or inadequately defined. An inadequate requirements analysis phase will result in more changes to software requirements during later development phases. The later in the development cycle the change occurs, the higher the cost due to increased rework. A mature software development organization may be able to account for these requirements changes in initial estimates based on historical data from previous projects. For example: if the company knows that in the past it has experienced a 1% per month change in requirements during development, it can take this into account in developing future estimates.

Historically, software programs experience a 1% - 5% change in requirements per month from the completion of the requirement analysis until the beginning of systems integration testing. Many systems continue these requirements changes during system testing, which is even more disruptive and costly. The US average for both commercial and military software is approximately 2% [2] requirements change per month between the requirements analysis and test phases. Thus, the average software development effort will experience a 24% change in requirements per year during the design and coding phases. Additionally, defects caused by poorly defined and contradictory requirements make up approximately 20% of all requirements but more than 30% of the most intractable and difficult to repair [3]. Testing is often ineffective in finding such requirements errors since the tests have been designed to determine if a requirement was implemented, not to determine if the requirement itself is correct.

Because of the impact requirements will have on the project's cost, schedule and performance, it is essential that requirements be carefully tracked and managed. It is also essential that a thorough, complete and well thought out analysis of the systems requirements be done. Not only during the software requirements analysis phase, but also during the development of the Operational Requirements Document (ORD), Mission Needs Statement (MNS) and System Requirements Specification (SRS). Use of Joint Application Design (JAD) and Quality Function Deployment (QFD) have both been shown

[2] Estimating Software Costs, T. Capers Jones, McGraw Hill, 1998, pp25, 40, 148, 193

[3] Estimating Software Costs, T. Capers Jones, McGraw Hill, 1998, pp423

to be highly effective in ensuring system and software requirements are well defined[4] and in reducing subsequent requirements change rates.

2.2.2 Description

Software requirements will be tracked from the System Requirements Specification (SRS), and Software Requirements Description (SRD) (or equivalent document if developed under a standard other than reference (b) IEEE/EIA 12207). The number of requirement changes over the development life cycle should also be tracked for each of these documents. The number of requirements and changes to requirements should be collected and reported for each Computer Software Configuration Item (CSCI) of each build.

2.2.3 Data Collection

Agreement by the developer on the question "What is a requirement?" is critical. Are requirements only delineated by the word "shall"? Is the statement "The developer shall provide: (a) ..., (b) ..., and (c) ..." to be considered as one or three requirements? If the specification states "... provide a display for terrain following.", does an unspoken derived requirement to provide range circles for threat Surface-to-Air Missiles result?

2.2.3.1 Source

The developing agency should provide software requirement data as derived from the new or revised SRS, or SRD.

2.2.3.2 Frequency

The initial number of requirements should be established at the milestone reviews as shown in Table 2-1. The developer should provide monthly reports after initial values are established.

TABLE 2-1. Establishment of Initial Requirements.

Source Document	Review Event
System Requirements Specification (SRS)	System-level requirements milestone review – System/Subsystem Requirements Review
Software Requirements Description (SRD)	Software-level requirements, milestone review - Software Requirements Review

Under the spiral and incremental development methods, it is possible for the System/Subsystem Requirements Review and especially the Software Requirements Review to occur more than once. Such a review could be required for every software spiral, increment or build in the program depending

[4] Estimating Software Costs, T. Capers Jones, McGraw Hill, 1998, pp426, 429

on how the program is structured. This being the case, requirements analysis cannot be viewed as it is in a waterfall type development, as something that occurs solely at the beginning of the development, but as a task that overlaps with other phases for other spirals/increments/builds. Thus more insight into the effectiveness of the requirements analysis on a software development will be achieved if requirements changes are analyzed for the individual spirals/increments/builds. This allows the analysts to determine if there is a significant amount of changes in requirements occurring following the requirements analysis phase for an individual spiral/increment/build. If the requirements changes are viewed only as totals for all spirals/increments/builds, it will be difficult or impossible to differentiate between expected requirements growth during the planned requirements analysis phases, and growth in later development phases due to inadequate up front analysis or lack of control of new user requirements.

2.2.4 Requirements Metrics

2.2.4.1 Requirements Quantity

2.2.4.1.1 Purpose

It is critical that a program have an accurate running account of the requirements of the system. Changes in the number of requirements are a strong indicator that other changes will occur in the program, such as size, schedule, cost, capacity, etc.. Requirements drive every aspect of a software development.

Note: The government should also have a requirements tracking metric in place for the Operational Requirements Document (ORD), Mission Needs Statement (MNS) and other government generated specifications.

2.2.4.1.2 Description

This metric shows the total number of requirements in top-level documents, the System Requirements Specification (SRS) and the derived requirements found in Software Requirements Description (SRD). The requirements data for this metric come from the software developer. The requirements data is graphed over time for the requirements found in the SRS, and each SRD.

Consideration: The requirements at all levels of the system: ORD, MNS, SRS and SRD should be tracked, traced, and modeled using automated tools (Doors, ReQuire, etc.). It would not be unusual to have thousands or even tens of thousands of requirements in a program. These cannot be adequately monitored by hand, and automated tools are therefore essential.

2.2.4.1.3 SRS and SRD Requirements Quantity Analysis Example

Figure 2-1 shows a sample chart of requirements trends. Some of the programmatic issues brought to light by this example are:

1. Notice that during both the systems and software requirements analysis phase, the number of requirements increases rapidly. This is to be expected since this is the period during which the requirements making up the SRS and the SRD are being defined.

2. Notice that the Systems Requirements definition is running concurrently with the Software Requirements definition between Feb & Jun at the beginning of the chart. Additionally, most of the systems level requirements are being defined after the Systems Requirements Review. This level of concurrency can lead to large numbers of errors and rework in the software requirements since they are being defined before most of the systems requirements they are derived from have been defined. These errors can continue to cascade into later phases such as design and code, which will raise the cost of rework much higher. The level of concurrency between systems and software requirements definition shown here is very risky and likely to cause large amounts of rework, which will have severe impacts on the project's cost and schedule.
3. It appears that the software requirements analysis phase ends in approximately Jun. Notice how total software requirements in Jun of about 660 continue to rise until Feb of the following year where they reach approximately 750. This represents about a 15% total increase or slightly less than 2% a month. This corresponds to US averages for software requirements changes. However, notice the changes are continuing into Developmental Test (DT). Requirements changes, with the associated rework involved will become more and more expensive as they occur later in the program. If the program had not planned for such increases over its life cycle, one would expect an even more rapid rise in cost to occur as these requirements continue to be added at later and later points in the development. Increases in SLOC and schedule slips associated with these requirements increases should also be expected.
4. Notice that approximately 30 or 40 new systems requirements are added at the beginning of DT. This will have a significant impact on cost and schedule for the program. Significant amounts of requirements analysis, design, coding and testing rework will be required to implement these requirements. Also notice that this abrupt increase does not cause a corresponding increase in growth in the software requirements. This may indicate that most or all of these systems requirements do not impact software, or it may indicate that the developer has not updated the software requirements have not yet been updated. Adding system requirements late in the development is extremely risky to a software development effort and is likely to cause large increases in cost and schedule.
5. When reviewing this metric chart, other metrics which should also be reviewed are:
    a. Size – Has the software size (SLOC, FP, etc.) increased as the number of requirements increased? If not why not? A mature developer may have adjusted the size estimate based on a level of requirement growth previously experienced. If this is the case, the developer should be able to show how the adjustment was made and its historical basis. Alternatively the size and therefore the cost and schedule estimates may not have been updated to account for requirements growth.
    b. Schedule – It would be expected that new requirements would have a schedule impact. A mature contractor may have planned for some predicted level of requirements growth, which could explain no change in schedule. Or as stated previously, maybe the developer is not keeping up with the requirements changes.
    c. Staffing – More requirements could require more staff.

d. Capacity – The more requirements, the more processing power, RAM and IO could be required to execute the code required to implement the requirements.

e. Cost – Expect cost to increase as the number of requirements increase.

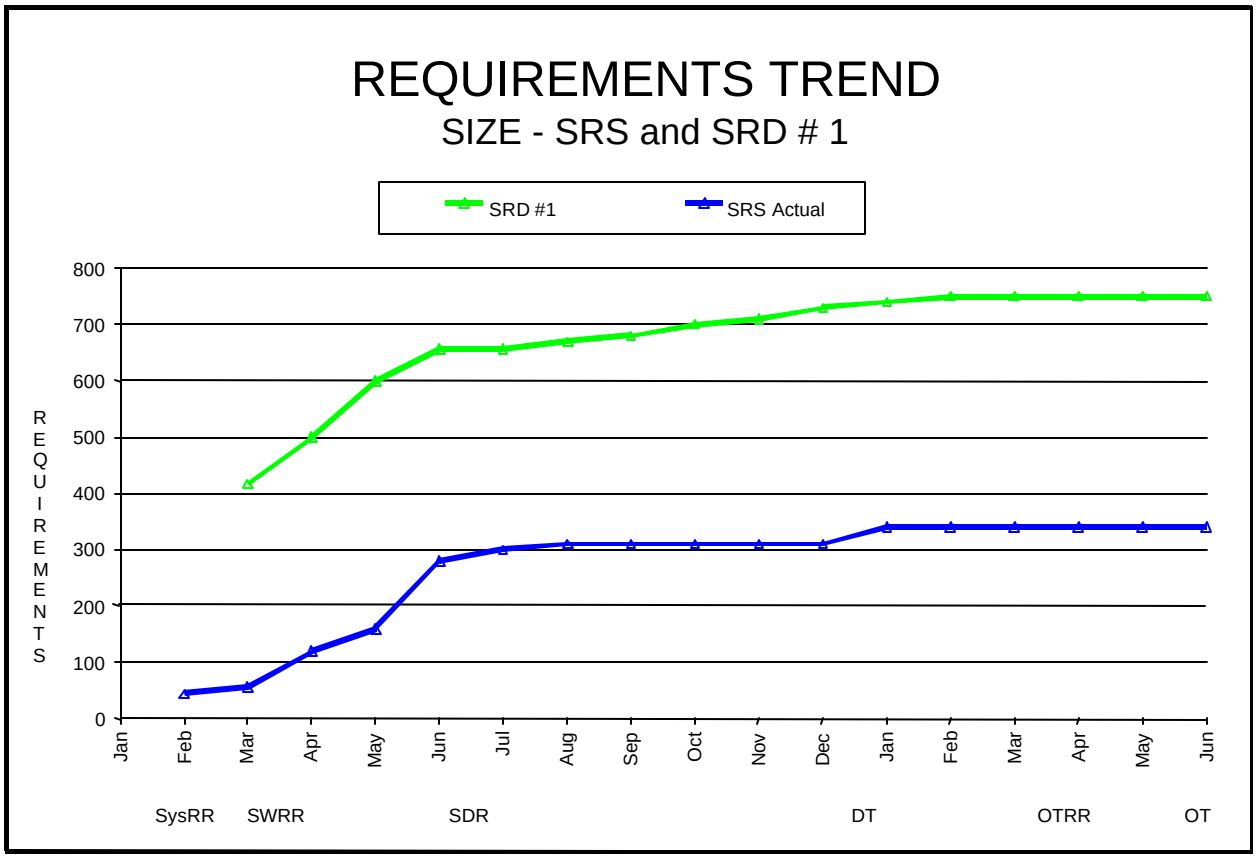


FIGURE 2-1. Requirements Trends for System Requirements Specification, and Software Requirements Description # 1.

2.2.4.2 Requirements Stability

2.2.4.2.1 Purpose

While total number of requirements can be used as a gross indicator of resource requirements, monitoring rates of change is also essential. The Requirements Stability metric provides the details of rates of change over the specified intervals. Even if total number of requirements stays constant, requirements deletions, additions and modifications can drive cost and schedule higher.

2.2.4.2.2 Description

This metric shows the added, changed, and deleted requirements in any particular document (i.e., SRS, SRD) for a specified interval of time.

The requirements data for this metric come from the software developer, and are depicted in a simple graph over time of the added, changed, and deleted requirements for individual documents.

Consideration:	*Agreement by the development group on the question "What is a requirement?" is critical. Are requirements only delineated by "shall"? Is the statement "The contractor shall: (a)..., (b)..., (c)..." one requirement or three requirements? If the specification says "...provide a display for terrain following" for example, is an unspoken derived requirement to provide range circles for threat Surface to Air Missile (SAM) sites included as a requirement?*

2.2.4.2.3 Analysis

While a global picture can be obtained from the total requirements information, the "devil is in the details" when requirements are changed, each change must be analyzed to understand the perturbations on the total program resources.

Note:	*The program must consider establishing a committee to review all requirement changes. The significance of this statement is that merely replacing one requirement with another is no assurance that the resources required for the changed requirements will remain constant.*

2.2.4.2.4 Requirements Stability Chart Analysis Example

Figure 2-2 provides an example of a requirements stability metric. Notice the following:

1. After peaking in Jun, the total number of requirements decreases, than increases, and decreases again. Keep in mind that all of these changes will have impacts on the cost and schedule along with other metrics such as size.
2. The total number of requirements drops from approximately 260 in June to 200 by the end of the program, about 23% total or a little less than 2% per month when considered through the end of the program.
3. However, when all the changes and deletions are considered over this twelve-month period we find a total of almost 400 changes, deletions and modifications to the requirements have been made. Approximately 225 of these have occurred after the system entered DT. Thus, the actual change rate for requirements on this program is 158%, or almost 13% per month. We would expect large cost and schedule overruns on this program due to the extremely high rate of requirements change.
4. Notice that the software requirements analysis phase appears to have continued at least through June, based on the steep increase in requirements up to that point. However, the Software Requirements Review was done in April, when less than half of the requirements were defined.
5. We should be asking the following questions about this project:
    a. What was the cause of the large number of requirements changes during DT?

b. Did changing user requirements cause this? If so, the large number of modifications should have resulted in a completely revised budget and schedule for the program with OPEVAL delayed until the associated rework could be completed.

c. Was it caused by the developer not discovering until DT that they had misinterpreted virtually all the requirements? Keep in mind, more changes have been made during DT than the total number of requirements in the system, thus some of the requirements must have been changed multiple times.

6. Why did earlier test phases not disclose this problem? Looking just at this metric, it is difficult to determine just what went wrong with this project. Review of other metrics relating to test coverage and quality may have shown that earlier testing was inadequate and that the system entered DT too soon. It can also be concluded from this situation that the requirements analysis phase for this project was inadequate. The large number of requirements changes may have been driven by other interfacing systems or user requirements. If this is a risk, requirements changes from these outside sources should be separately tracked and the and corrective action taken.

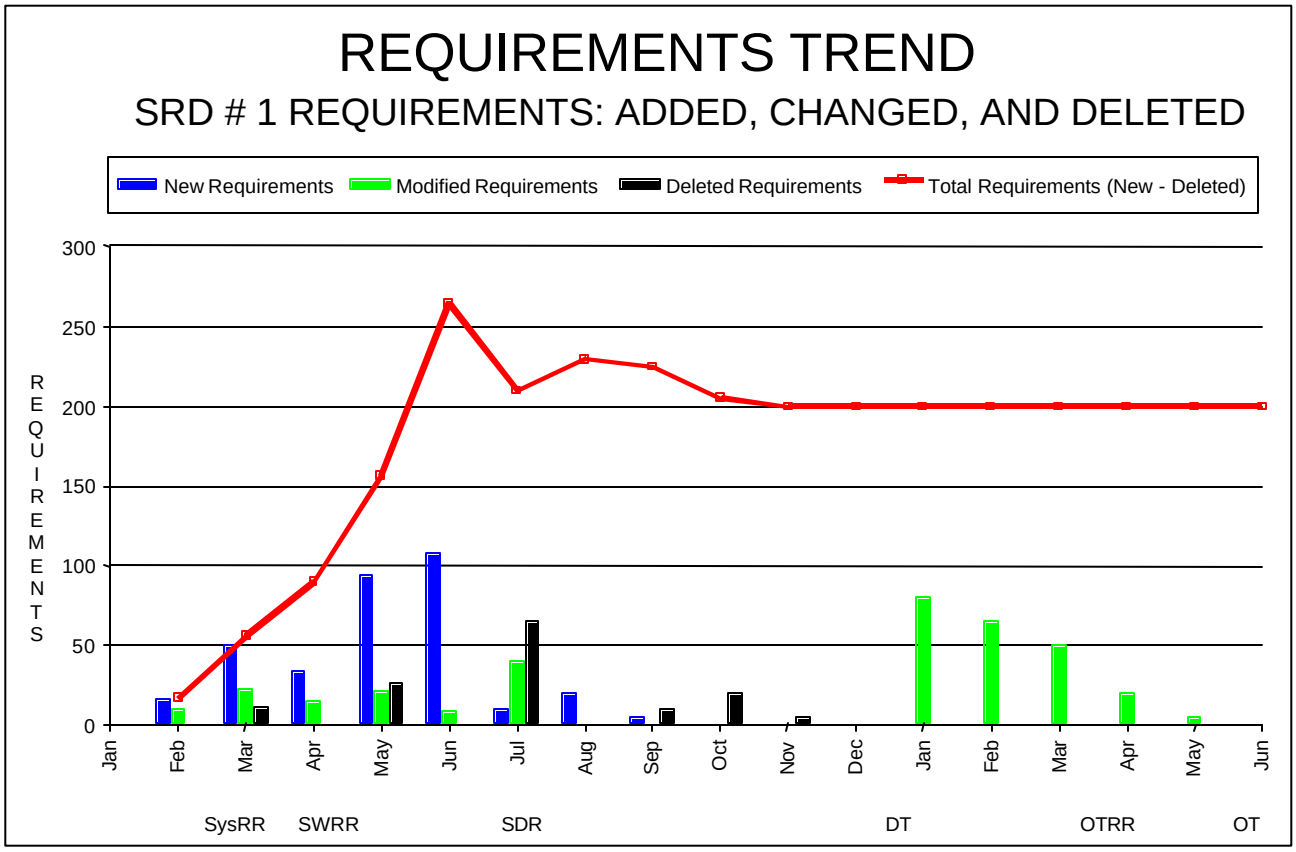


FIGURE 2-2. System Requirements Specification: Added, Changed and Deleted.

2.2.4.3 Requirements Test Coverage and Tests Completed

2.2.4.3.1 Purpose

Each requirement should be testable and be tested. This metric tracks the progress in both developing test procedures for the different test phases and successful test completion.

2.2.4.3.2 Description

The Requirements Test Coverage and Tests Completed metric shows the number of requirements for which test procedures have been written and the number of requirements that have been tested compared to the total number of requirements. Additionally, it must include not only the actual rate at which test cases are being developed and executed but how they compare to the planned rates for developing and completing tests. Tests are not completed until they have been successfully passed.

2.2.4.3.3 Analysis

The program's software development schedule should be used as the basis for the planned test case development and the planned test case completion. This plan needs to be updated as the total number of requirements requiring testing changes during development. A mature software developer would be expected to allow for some amount of requirements changes, deletions and additions based on historical data from previous projects. If this is not done, it is virtually certain that the project will not meet its test schedule.

2.2.4.3.4 Requirement Test Coverage Chart Analysis Example

Figure 2-3 provides an example of the Test Coverage and Test Completed Metric. The sample shows planned goals for test procedures developed and tested leading up to Software Qualification Testing for the software item in SRD #1. A few questions and observations we can draw from this metric are:

1. While the rate at which test descriptions are being developed is less than planned, the rate is adequate so that the planned test completion schedule does not appear to be impacted. If we look at May 02, we see that over 300 more test procedures are available, than are planned for execution in that month. It is unlikely that a lack of test descriptions is restricting the developer's ability to run tests based on this data.
2. Test completion is lagging approximately a month behind the planned rate. Why?
    a. Are more tests failing than were allowed for? This would indicate a quality problem. Quality metrics need to be investigated in this case.
    b. It appears that testing started a month late. Why?

        i) Perhaps the code was not yet ready for testing. The program schedule and size metrics may help to determine if this is the case.

        ii) Perhaps there is a staffing shortfall preventing testing execution. Check staffing metrics to see if this is a possibility.

    c. Is the SQT testing on the project's critical path? Will this delay cause other slips in the schedule? Review the program's project schedule to determine if this is the case.

3. Keep in mind that changes, deletions and additions to requirements will also impact the test schedule. In our example this is not an issue. If it was an issue, the following problems would arise.

   a. New and modified requirements will force additional test procedures to be developed or existing procedures to be modified and executed. Modifications to requirements that have been completed will require the modified test procedures to be rerun. Other test procedures may also need to be rerun to ensure any changes to the code have not caused problems in other areas.

   b. Deleted requirements may necessitate the rerunning of test procedures to ensure the code modifications have not caused problems in other areas. The later the requirement is deleted, the more retesting will be required.

   c. Schedule and cost impacts are very likely. The later a requirement is added, deleted or changed, the more rework in testing and other areas will be required to implement the modification.

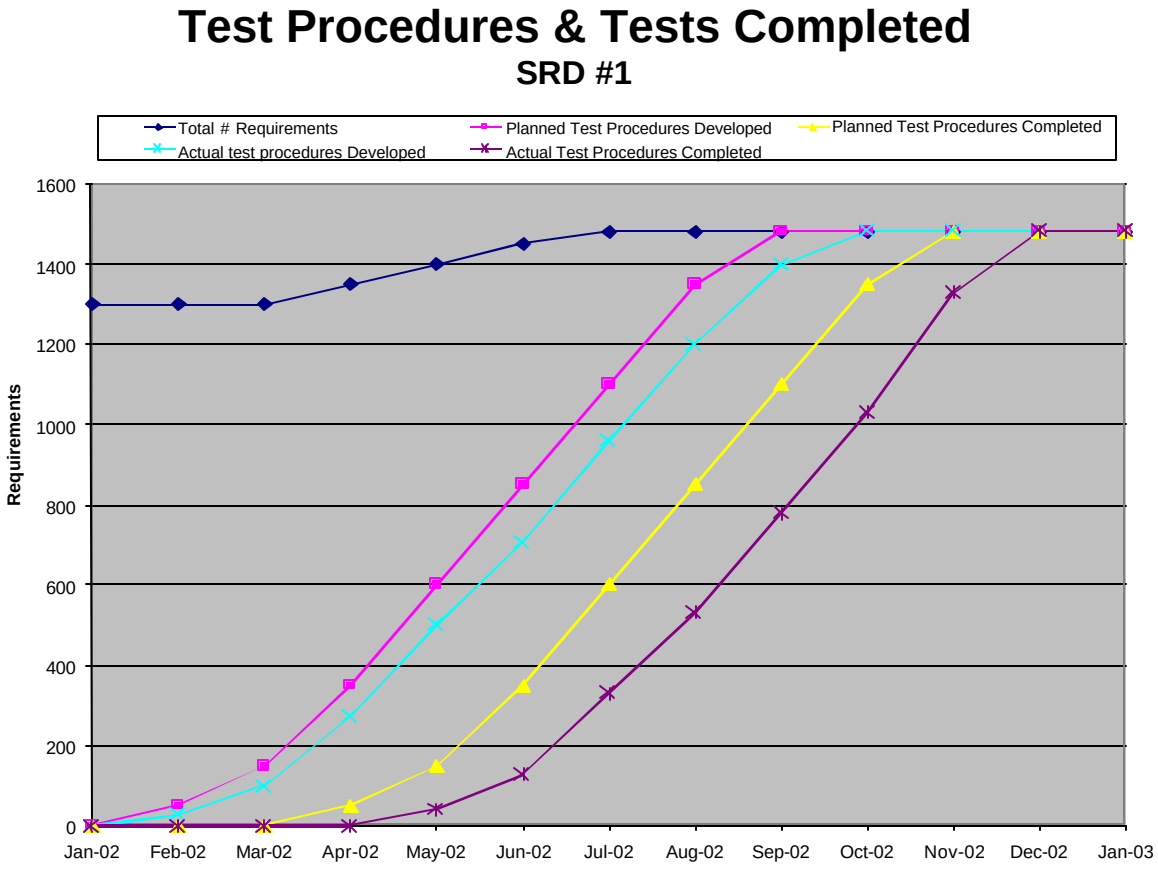


FIGURE 2-3. Test Procedures and Tests Completed.

2.2.5 Requirements Earned Value Management (EVM) Issues

The primary objective of a software development from the customer's point of view is the implementation of the desired requirements within the planned cost and schedule. Thus, wherever

possible, earned value should be allocated based on the implementation of requirements[5] for the development phase in question[6]. For example:

1. For the design phase, earned value credit should be taken when the design for a requirement is completed. It doesn't matter how many pages of documentation have been produced if the design isn't complete. Thus, taking earned value in the design phase based on level of effort, documentation produced, or some other metric does not necessarily reflect the actual status of the effort.
2. In code and unit test, earned value credit should be taken when the requirement has been coded, passed peer review and unit tested. It doesn't make any difference how much SLOC is produced if it doesn't implement the requirements. Thus, basing earned value on SLOC or some other metric will not necessarily accurately reflect the project status. What is important is if the requirement has been implemented.
3. During testing (other than unit testing), earned value should be taken when the code for a requirement is successfully tested. Taking credit based on the number of test cases run will not necessarily indicate if the requirements have been successfully tested.

If requirements are to be used as the basis for earned value, an effective method of tracking system requirements through intermediary software requirement, design, code and test phases must be implemented. Such traceability is essential in any case if it is to be determined which of the systems requirements have actually been implemented. This essential traceability also can be used to increase the effectiveness of software earned value for the system.

2.2.6 Requirements Measurement & Metrics References

Further information on requirements measurements and metrics is available in Reference (c) "Practical Software Measurement, Objective Information for Decision Makers", pages 176 – 179.

2.3 SIZE

2.3.1 Purpose

Software size (SLOC, FP, etc.) is the primary driver of total software development cost and schedule. The software size estimates and the actual size will be determined by the planned requirements and what requirements were actually implemented in the final product. Since software size estimates are often very inaccurate during early phases of a project, when its budget is being developed, errors in the software size, which in the vast majority of cases are too low, are the major cause of cost and schedule overruns. Because of its impact on cost and schedule, it is essential to any program that the size be tracked in order to spot indications of size growth and take corrective action as early in the program as possible.

---

[5] "Practical Software Measurement, Performance-Based Earned Value" http://www.testablerequirements.com/Articles/solomon.htm, http://www.stsc.hill.af.mil/CrossTalk/2001/sep/solomon.pdf, CrossTalk, September 2001, Paul Solomon, Northrop Grumman Corporation

[6] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

2.3.2 Data Collection

2.3.2.1 Source

The software developer will be responsible for providing all size estimates and actuals. Actual size should be taken from source code that has been compiled and is, preferably, under configuration management.

2.3.2.2 Frequency

Report size estimates monthly from contract award through completion of development and testing. Report actual size monthly from the time code is placed under configuration management until the completion of system test.

2.3.2.3 Size Attributes

Software size for estimates and actuals should be broken down based upon the following attributes:

1. New code – Size of new code. Generally the most expensive to develop. Often underestimated.
2. Reused Unmodified code – Size of code being reused unmodified. This code meets some allocated set of program requirements without modification. Still requires integration testing with new and reused modified code to ensure deleted code did not cause any defects. Functionality that can be implemented with reused code is often overestimated. Additional new and modified code is than required to make up the difference.
3. Reused Modified code – Size of code that is being modified prior to reuse. This code will consist of modules, classes, functions, routines, Computer Software Units (CSU), Computer Software Component (CSC) and CSCIs, which must be modified to some extent to meet the program requirement. The higher the percentage of code that must be modified, the more expensive the modifications. In some situations, where the code is not well documented, poorly written and/or has numerous defects, the cost of modifying the code can exceed the cost of developing it from scratch. This is especially true for code that was not originally designed for reuse, or the development team is unfamiliar with the code.
4. Deleted code – Size of code to be deleted prior to reuse. This is code that must be deleted from code being reused since it meets no system requirements. Deletion of code can be very expensive. Deletion requires the developer to determine what functionality must be removed, determine what code in what modules to delete and then test afterwards to determine if the deletion caused any unexpected defects. For code not designed for reuse, is poorly designed, of low quality, and/or poorly documented, deletion can be very difficult and expensive due to the effort involved in determining what to delete and what associated modifications and new code are required to prevent the deletions from causing defects.
5. Automatically Generated code – Size of code to be produced by automatic code generators. Even though the source code is automatically generated, requirements analysis, design and testing must still be performed. Thus, the use of automatic code generators may eliminate less than 20% of the

effort compared to manual generation. Automatic generation is generally limited to very specific areas such as user interface development. It will not eliminate all manual code generation.

6. Language in which the code is developed. Useful for tracking performance and quality in developing software in various languages.
7. Spiral/Build/Increment/Release in which the code is developed. Very useful for comparisons of the amount of code actually developed in initial spirals, builds, increments or releases as compared to the estimates. If the actual code is significantly different than the estimates, the estimates for later, spirals, builds, increments or releases must be updated along with the cost and schedule.
8. CSCI or software item in which the code is included. Different components of code vary in complexity. Breaking down the code between CSCIs or software items helps to determine where the estimates vary the most from the actuals. This allows the targeting of resources to correct problems in the specific modules that are having problems.

The eight previously identified code attributes all take different amounts of effort to implement, develop and/or integrate into the system. These different attributes should thus be broken out in order to identify how changes can affect the cost and development schedule. Not only are initial size estimates often low, but developers also often underestimate the amount of new code required (the most expensive type of code), while overestimating the amount of reused unchanged, reused modified and automatically generated code (less expensive types of code). Thus, the total amount of code can stay relatively constant, but if the amount of new code increases while reused and automatically generated decreases, the cost of the system and the schedule to complete will increase.

Tracking the different types of code also allows the amount of effort and time to implement, develop and integrate the code to be tracked. This data is extremely valuable for use in verifying that the developers planned productivity is being achieved, for updating the estimate to reflect changes in requirements and other metrics, and for providing the historical basis of actual productivity necessary to develop estimates for future systems. See the discussion of staffing metrics for further information.

2.3.3 Size Metrics

2.3.3.1 Source Lines of Code (SLOC)

2.3.3.1.1 Description

There are a variety of means by which SLOC can be counted. The two major methods are counting physical and logical lines of code. Physical lines of code can be thought of as counting each individual line (carriage return) while logical lines of code attempts to count only executable lines. The Software Engineering Institute (SEI) document entitled, Software Size Measurement: A Framework for Counting Source Statements, reference (e), provides additional guidance on the SEI methodology for counting physical and logical lines of code. Keep in mind that the counting details used by different software development organizations vary. It is therefore essential that the counting method used by the developer be understood in order to compare software sizes from different organizations.

Code Counting Programs (CCPs) should be used if they exist for the software language used in the development. Contact the Naval Air System Command (NAVAIR) Software Engineering Division

(AIR-4.1.11) or SRC for information on available CCPs. Use of such CCPs is also useful in ensuring a meaningful comparison of application sizes from different developers since it ensures the counts are done in a consistent manner.

2.3.3.1.2 Purpose

To estimate CSCI and overall program size.

2.3.3.1.3 Description

This metric shows the size of spirals/builds/releases/increments in both new, unmodified reused, modified reused, deleted and automatically generated SLOC. A program will also want to develop SLOC graphs for individual CSCIs and/or software items.

The data for the SLOC metric comes from the software developer, and should be shown for each build of the software. Individual software items can also be tracked to provide detail.

Consideration:	*The question "What constitutes a Line Of Code?" must be clearly defined for both the contractor and subcontractors for this metric to have any meaning.*

2.3.3.1.4 Analysis

In today's software development environment, spiral development, multiple builds and incremental release methodologies are often used. Each spiral, build or release builds on the functionality and developed in previous spirals, builds or releases. This means that successive spirals/builds/releases will contain the same code as their predecessors. Thus, the total cumulative size of the software is the size of the last spiral/build/release. As requirements for the project are changed, deleted and added, the SLOC estimate must be updated.

2.3.3.1.5 Rules of Thumb

Major variations in the size data could indicate:

1. problems in the use, appropriateness, or validity of the model used to develop the estimates,
2. instability in requirements, design, or coding,
3. problems in understanding the system to be developed, and
4. an unrealistic original estimate for the system to be developed.

2.3.3.1.6 SLOC Implementation Chart Analysis Example

Figure 2-4 shows an example of a SLOC implementation metric chart. The planned lines on the chart show the estimates for the different types of SLOC during the coding phase. The initial parts of the planned lines show the expected rates at which the code will be developed, integrated or deleted during this period. The actual lines show the amount of the different types of code actually developed, integrated or deleted during this phase. This easily allows a comparison of how the software development is actually proceeding in comparison to the program plan. In order to simplify this chart,

automatically generated code was not included. For our purposes we will show only actuals and the initial estimate from the start of the code and unit test phase.

1. Notice that the plan calls for development to be completed by Jul 02 for all types of code. However, it can be seen that the actual development effort continued to Nov 02 due to a combination of lower than expected productivity and higher than planned SLOC. This four-month slip may or may not impact the overall program schedule depending on whether the software development is on the programs critical path.
2. The actual new SLOC is considerably higher than the planned new SLOC. Why is this the case?
    a. Was the original estimate inaccurate? Look at the requirements metrics, if there have been no changes in the requirements, or the amount of changes is not large enough to account for the SLOC increase, than at least part of the problem was an inaccurate initial estimate.
    b. Was the increase caused by additional requirements? If the SLOC count has increased partially due to additional requirements, than the customer maybe getting additional functionality in partial compensation for the increased cost and schedule required for the increased SLOC.
    c. While the actual new, reused modified and deleted SLOC has increased, the amount of unmodified reused SLOC has decreased. This would seem to indicate that the amount of requirements that could be implemented in reused code was overestimated and more new and modified code was required as an alternative. This may also be a result of changes in requirements that reduced the amount of code that could be reused. Review of requirements metrics should give some indication if this was the case.
3. While it can be seen that the actual production rate for new code is less than planned, this metric does not provide an indication that the total amount of SLOC will increase beyond the planned level prior to Jul 02. To determine if this would be the case, prior to Jul 02 other metrics must also be evaluated. Assume that we are taking earned value based on the implementation of requirements during the software development phase, as discussed in section 2.2.5. Also assume that there are no requirements changes. For the sake of this discussion, ignore the chart after Jul 02 in figure 2-4. There are several different alternatives a comparison of EVM and the SLOC metric could indicate:
    a. What if the earned value indicated that the program was on or ahead on cost and schedule, Cost Performance Index (CPI) and Schedule Performance Index (SPI) are greater than or equal to 1? This would indicate that the requirements were being implemented at or faster than the planned rate. The developer has overestimated the amount of code required to implement requirements. Even though the code production rate is less than planned, the program is on or ahead of schedule because of this implementation rate. The expected result would be that the total amount of new SLOC would be less than planned. This is an unlikely scenario since the amount of SLOC is rarely overestimated. If this situation does occur, the EVM should be carefully reviewed to insure it is providing an accurate evaluation of the status of the program.
    b. If CPI is one, and SPI is less than one, the project will be on cost behind schedule. SLOC estimate should be close to planned. Cost to implement requirements agrees with plan, but implementation is slower than expected. This sounds like there may be staffing problems with not enough people working on the effort.

c. CPI and SPI are less than one. Its costing more than expected to implement requirements and its taking longer than planned to do so. This sounds like the SLOC will probably increase, although the unfavorable CPI could also be caused by higher than expected labor rates. Another possible staffing problem? Is the developer being forced to pay more than expected for software developers? Is the schedule delay because enough qualified staff can't be found?

d. CPI less than one, SPI equal to one. Project is on schedule, but cost to implement requirements is greater than expected. Again this could either indicate that SLOC will increase or that staffing costs are higher than expected, either due to a need for more people than expected to get the work done, or higher labor rates than expected, possibly both.

4. Any method that seeks to predict if the amount of SLOC will increase beyond the planned level must make a comparison of the actual amount of SLOC completed to the actual requirements implemented. This is what the previous example using EVM attempts to do. Figure 2-5 shows planned and actual implementation of the requirements to be implemented in the new code in Figure 2-4. For the purpose of our analysis, we will assume that it takes roughly the same amount of effort to implement each software requirement. For detailed software requirements as documented in the SRDs it should take approximately the same effort to implement individual requirements. SRD level requirements roughly correspond to Testable Requirements[7].

   a. Notice in Figure 2-4, that in Apr 02, approximately 71% of the planned SLOC has been completed. If we review Figure 2-5, we see that in Apr 02, approximately 53% of software requirements are implemented. This would seem to indicate that the SLOC estimate is too low since it appears to be taking about a third more SLOC to implement a requirement than was planned. We see similar differences for the other months proceeding Jul 02 in Figure 2-4 & 2-5. This would seem to indicate that the final SLOC would be approximately a third greater than the planned new SLOC of 50,000.

   b. If more than 71% of the planned software requirements had been implemented by Apr, this would have indicated that the SLOC estimate was too high, since it was taking less code than predicted to implement each requirement.

   c. Keep in mind in the real world there is likely to be more than one reason, for SLOC growth; low initial estimate, software requirements changes, systems requirements changes, etc..

[7] Sizing Software Using Testable Requirements, http://www.testablerequirements.com/

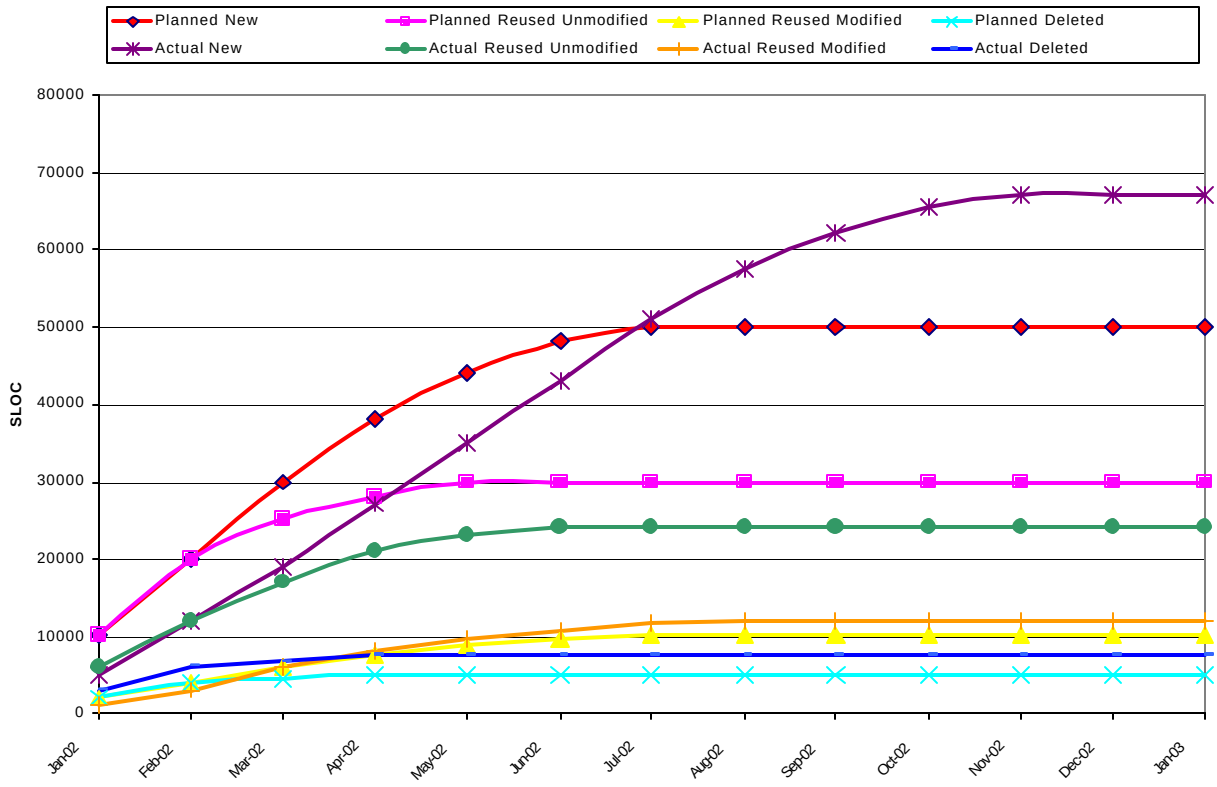


FIGURE 2-4. Software Size Implementation Trend, SLOC.

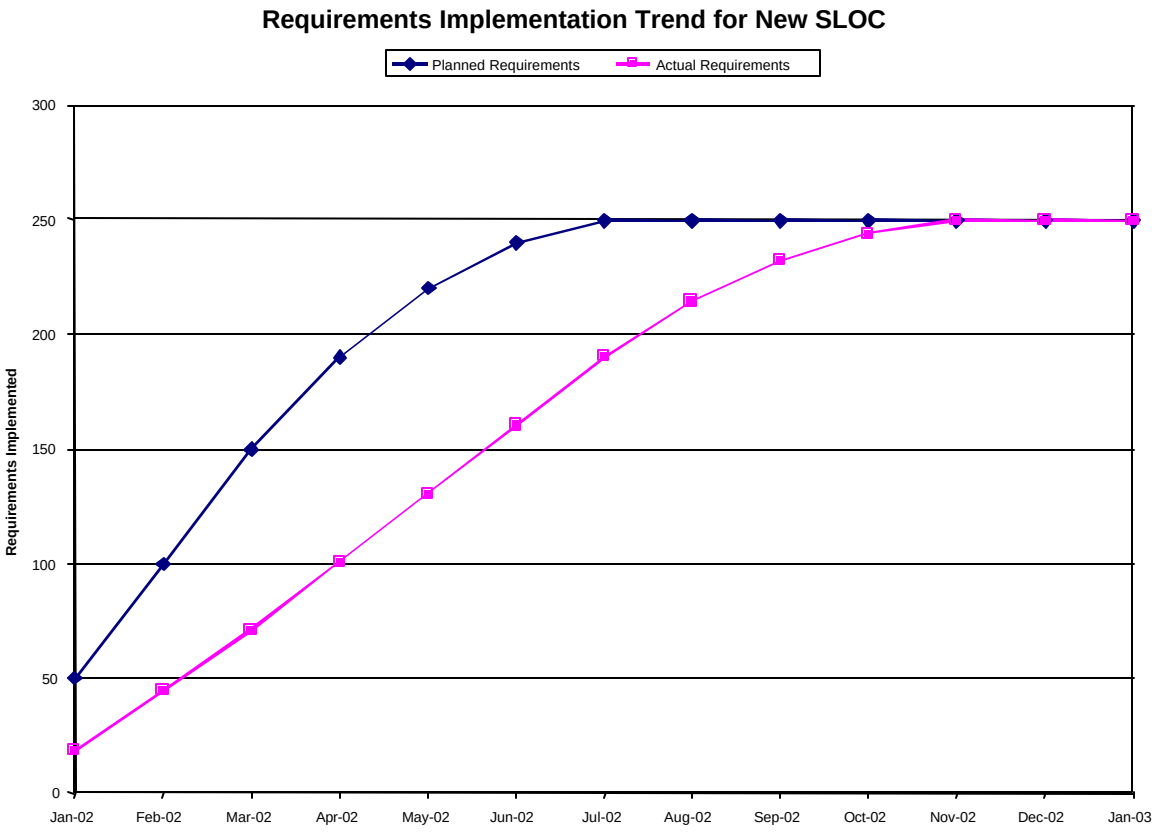


FIGURE 2-5. Requirements Implementation Trend for New SLOC

2.3.3.1.7 SLOC EVM Issues

One method of taking earned value credit is to use the amount of SLOC completed[8]. When using SLOC for earned value, consider the following:

1. Using SLOC for earned value is only appropriate in the code and unit test phase.
2. SLOC estimates are often low and will tend to climb during development. This will result in the CPI and SPI being too high if a low SLOC estimate is being used as the basis of earned value. SLOC estimates must be updated at least monthly based on current data and earned value should be based on the % of the current SLOC estimate that is complete.
3. The definition of when the code is completed must be established if SLOC is to be the basis of earned value. Completion of peer reviews and/or unit test is often used as completion criteria.

2.3.3.2 Function Points (FP)

2.3.3.2.1 Description

Function Points are a software sizing metric based on the analysis of the software requirements. A function point is essentially a standard unit of software functionality. The function point counting rules define how to determine how many of these units are in a specific software application. The most widely utilized function point counting rules are those established by the International Function Point Users Group (IFPUG), www.ifpug.org. IFPUG provides multiple sources for training in function point counting, certifies function point counters, and acts as a source for companies providing function point counting services. Estimation of software size using SLOC requires extensive experience in the domain of the application to be developed along with supporting historical data from similar applications to support the estimate. FPs on the other hand rely on the use of trained function point counting experts to derive the FP size from the software requirements documents.

> *It is **ESSENTIAL** that the personnel performing the function point count have received formal training in Function Point counting, and at least one member of the team is a **CERTIFIED FUNCTION POINT SPECIALIST**.*

In cases where a size estimate is needed, but the historical data and/or domain knowledge necessary to perform a SLOC estimate is unavailable, or a well defined method of performing the SLOC estimate is unavailable, FPs provide a useful alternative. SLOC estimates are often generated via an adhoc and informal process with inadequate attention being paid to a careful analysis of the systems software requirements and historical data from similar systems. This is one of the primary causes of the low SLOC estimates and resulting poor cost and schedule estimates many systems are plagued by. Because FPs are based on a rigorously defined process of counting rules, based on analysis of the software requirements, FPs avoid the inaccuracies often caused by informal adhoc SLOC estimates.

While there are many advantages to FPs, they are not a cure all silver bullet for software sizing problems. As with any software process it will take time and effort to adapt a software development

[8] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

environment to use FPs. If the organization's current metrics are based on SLOC or some other size metric, further effort will be required to calibrate or normalize this data to a FP sizing methodology.

Function Point counting rules are based on an analysis of the software requirements to count the following components of the software:

1. Internal Logical Files (ILF) – A user identifiable group of logically related data or control information maintained within the boundary of the application.
2. External Interface Files (ELF) – A user identifiable group of logically related data or control information referenced by the application, but maintained within the boundary of another application.
3. External Input (EI) – An elementary transaction that processes data or control information that comes from outside the application boundary.
4. External Output (EO) – An elementary transaction that processes data or control information sent outside the application boundary.
5. External Inquiry (EQ) – An elementary transaction that sends data or control information outside the application boundary without processing – a simple request/response or "fetch."

For further information on counting rules, see the Function Point Counting Manual available at www.ifpug.org.

2.3.3.2.2 Analysis

Once a count has been completed, analysis of FPs is similar to analysis of SLOC. In a development consisting of multiple spirals, builds, releases or increments, each subsequent spirals/builds/releases will be built upon the functionality implemented in the predecessors. Changes, deletions and additions to software requirements will also require the function point count to be updated.

Charts of function point metrics will be similar to that shown for SLOC in figure 2-4. The primary difference is that the number of FPs will be less than SLOC, on average one FP is equivalent to approximately 55 SLOC of C++[9].

2.3.3.2.3 FP EVM Issues

FPs can also be used as a means of taking earned value[10]. Since FPs are closely related to software requirements, they can also be used to track earned value in all phases of the development after software requirements analysis. Some issues to consider:

1. Since a FP is a standard unit of software size and functionality, a standard amount of earned value and effort can be allocated to the completion of a FP.
2. The traceability for the system must also be able to trace the number of function points to different parts of the design, coding and unit testing, and integration testing. This is necessary so as part of

[9] "Applied Software Measurement" pages 80 – 92, Capers Jones, McGraw Hill, 1997

[10] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

the design is completed, or a unit of code is completed or a test procedure is completed, the number of function points worth of earned value to be taken can be determined.

3. Function point counts must be updated as software requirements change otherwise Earned Value based on them will be inaccurate.

2.3.4 Sizing Measurements & Metrics References

Further information on sizing measurements and metrics is available in Reference (c) "Practical Software Measurement, Objective Information for Decision Makers", pages 173 – 176.

2.4 STAFFING

2.4.1 Purpose

The Staffing metrics shows the relationship of planned versus actual staff hours to develop the software. Staff hours are tracked in the following software development phases; system level requirements design, software level requirements analysis, preliminary design, detailed design, code and unit test, component integration and test, program test, and system integration & test (software-to-software and software-to-hardware integration). The metric tracks the developing agency's ability to maintain planned levels of staffing. The measure includes engineering and management personnel directly involved with activities such as software system planning, requirements management & analysis, software design, code, test, documentation, configuration management, quality assurance, etc..

2.4.2 Description

The staff hours measure tracks hours expended by assigned software personnel and tracks the number of planned and actual software personnel, the number of software personnel losses, and the number of personnel additions for each reporting period. The data shows planned and actual staff hours over the development period. Regular and overtime hours should be reported separately. For current and future months, estimated hours (regular and overtime) will be reported. Actual staff hours (regular and overtime) should be reported for the previous months.

2.4.3 Data Collection

2.4.3.1 Source

The developing agency will provide staff hours data for efforts associated with; system level requirements design, software level requirements analysis, preliminary design, detailed design, code and unit test, component integration and test, program test, and system integration & test (software-to-software and software-to-hardware integration).

2.4.3.2 Frequency

An estimated staff hours profile should be presented at the inception of the project and should be updated monthly thereafter. The update will include previous month actuals and revised estimates (if required).

2.4.3.3 Format(s)

Ideally, staffing should be broken down by the following attributes:

1. Labor categories: management, systems engineer, requirements analyst, coder, tester, configuration management, quality assurance, documentation, etc.. Labor costs vary significantly for the different categories. This information is helpful in developing and updating cost estimates. Why are costs so high even though we have the planned number of people? Maybe the labor mix is different than planned; there may be more higher priced labor categories than originally planned.
2. Software development phases: system level requirements design, software level requirements analysis, preliminary design, detailed design, code and unit test, component integration and test, program test, and system integration & test, etc.. This is useful when attempting to determine why there are cost or schedule overruns occurring during one of these phases. Is staffing lower than planned? If staffing is at plan perhaps the assumed productivity for the phase is too high.
3. Spirals, builds, releases or increments, CSCIs. Many modern programs have overlapping spirals, builds, releases or increments all going on at once. Breaking staffing down allows it to be determined whether problems for that specific spiral, build, increment/release are staffing related.
4. Types of code: new, reused unmodified, reused modified, deleted, and automatically generated. The productivity for all these types of code will vary. If the mix of the different types of code being developed changes, than the staffing plan needs to change to accommodate it. If the amount of new code increases, while the amount of reused code increases, the overall result will be an increase in staffing since new code will have a lower productivity. Breaking down staff by the types of code they work on makes it easier to determine the impact of changes in the types of code to be developed.

These breakdowns allow the analysis of the metrics to determine what the causes of any issues are. The more this information is rolled up, the less visibility and ability management will have to determine the cause of the problem. Breakdowns at this level are also extremely useful in updating program costs and schedules to account for the current situation.

2.4.4 Staffing Metrics

2.4.4.1 Personnel and Staff Hours

2.4.4.1.1 Purpose

The number of personnel on the program provides information about one of the primary program resources. Staff hour information is used in conjunction with personnel to give a more precise "feel" for the human resources being expended on the program development.

2.4.4.1.2 Description

Software personnel in use on the program development can be graphed over time. Actual data should be compared to the developing agency's planned usage of personnel. Separate charts should be made for each subcontractor.

Consideration:	*The question "Who is a software person?" needs to be clearly defined to assure proper head counts are taken into account.*

2.4.4.1.3 Analysis

Programs "in trouble" must be monitored carefully to ensure that compensatory time, weekends and holidays, and unpaid hours do not artificially boost productivity. Whereas these techniques may work for short periods, heroic schedules over the long term are doomed to failure.

While a "head count" for personnel is provided in the previous metric, it is also important to know how much each "head" is contributing to the development effort.

Overtime with each "head" working 55-hour weeks cannot be expected to yield positive results after long periods.

Studies have shown that while moderate schedule pressure actually boosts productivity, increasing the schedule pressure results in reduced productivity due to burnout, higher defect rates and increased rework.

The reviewer should take into account "reality" information. While each 52-week year has 2080 hours, (assuming 40 hour per weeks), when sickness, holidays, and vacation are taken into account, the government shows that only 1761 regular hours [roughly] are available for actual work in a productive man year; 16% is already lost.

Consideration:	*Be sure to compare this staff hour information to the size and requirements metrics to assure consistency over time of the development effort.*

2.4.4.1.4 Rules of Thumb

There are two Rules of Thumb:

1. Significant under expenditure of hours may result from:
    a. difficulties staffing the contract,
    b. overestimating the software size, and
    c. increasing levels of open problems; and
2. Significant over expenditure of staff hours my result from:
    a. absorbing staff from projects that ended,
    b. underestimating the size of the software,
    c. increasing number of errors,
    d. changing requirements, and
    e. reduced reuse.

2.4.4.1.5 Personnel and Staff Hours Chart Analysis Example

Figure 2-6 provides an example of a personnel and staffing chart for an entire project. A few issues of interest about this example are:

1. Notice how in the early part of the project, staffing is running well below plan. Further if we compare staffing to staff hours, we notice that the number of hours per developer per month is running above plan. Some possible reasons for this deviation from planned staff levels:
   a. Required personnel are not available. Either the developer has been unable to hire them or other projects from which they planned to absorb staff are running over schedule. Review of other measurements and metrics should show the developer is behind in requirements analysis, design, coding, etc., if this is the case.
      i) EVM should be showing a SPI of less than 1.
   b. Developer has over estimated effort required on the project. This is unlikely given historical trends that show that the effort required for software developments is in the vast majority of cases underestimated. Evaluate other measurements & metrics for requirements, size, design progress, etc. to see if they also indicate the program has been oversized.
      i) EVM should be showing a SPI of one or more.
2. Notice that in May 02, even though the number of personnel is below the planned level, hours for the month exceeds the planned level. The developer is working the team harder to make up for a shortage of personnel. Excessive schedule pressure will result in reduced productivity due to reduced quality and additional rework required. Watch for signs of a reduction in quality due to the schedule pressure in the quality metrics.
3. From Jun 02 on the number of staff and staff hours exceeds the original plan. This could indicate the developer is attempting to make up schedule by throwing more bodies at the problem. However, the larger the software development team, the more management overhead and inter-team communication will be required and thus the lower the productivity per individual on the team. Even if the developer does manage to make up the schedule by using a larger team at this point, the result will probably be an increase in project cost.
   a. EVM in this situation would show an SPI less than one, which may be improving, but a CPI dropping in value.
   b. Review other metrics to see if they corroborate this trend. Keep in mind that low early staffing probably resulted in the requirements analysis and design phases running longer than expected. In order to make up schedule the developer must therefore reduce the time from what was originally planned for coding and testing. Additionally, there is a strong possibility of quality problems due to the low staffing early in the program. Thus the developer must make up for lost schedule while managing a larger less productive team and trying to correct for quality issues due to short cuts taken earlier in the program.

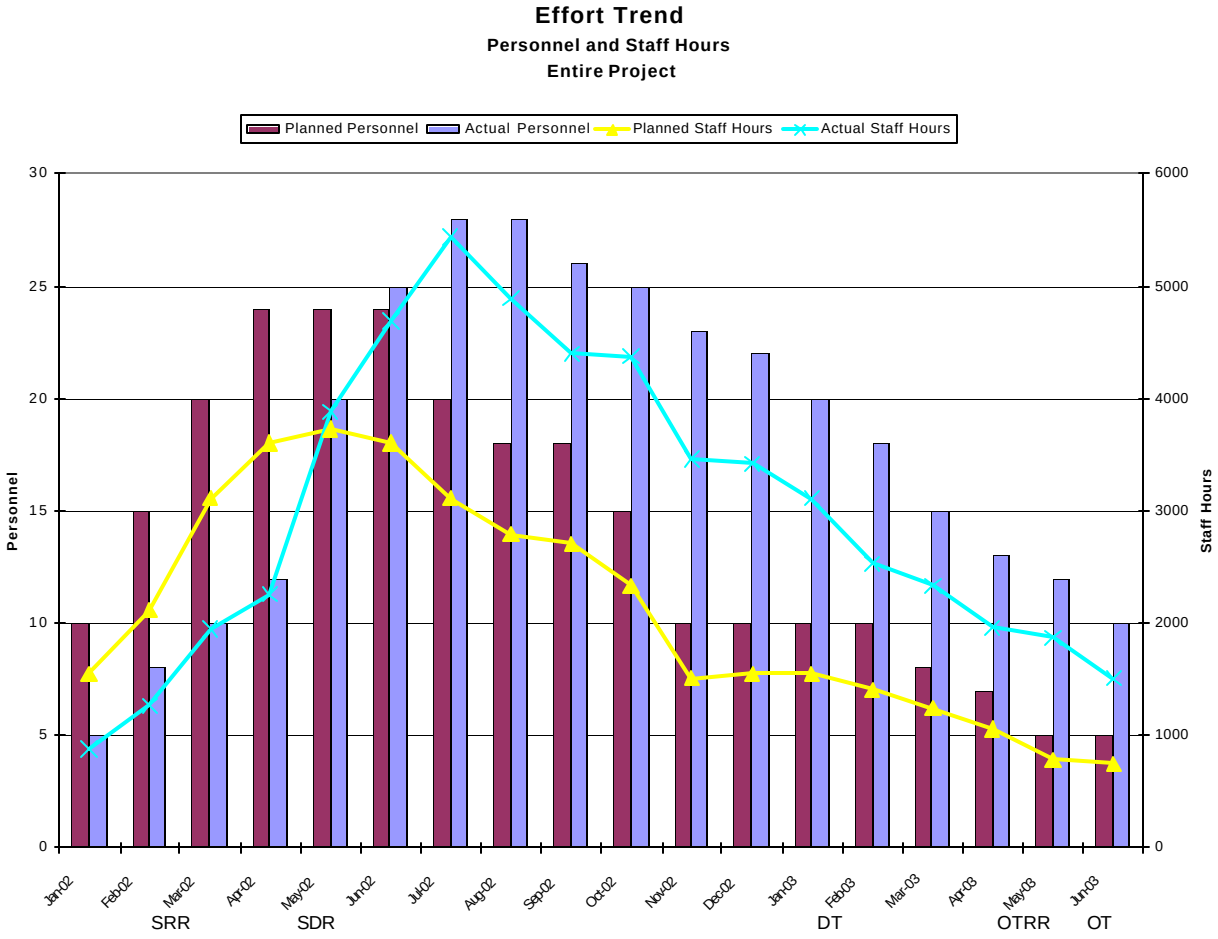


FIGURE 2-6. Personnel and Staff Hours.

2.4.4.1.6 Staffing EVM Issues

Using staff as the basis of EVM is called Level of Effort[11]. Use of Level of Effort should be minimized as much as possible since it gives no real indication of the progress in implementing the desired functionality in the software. Possible areas for use of level of effort may be in Project Management, Configuration Management, Software Quality Assurance, and Facilities Maintenance. However, even these areas may require greater staffing if the total size of the effort grows due to requirements increases or other reasons.

2.4.4.1.7 Staff Measurement and Metrics References

Further information on sizing measurements and metrics is available in Reference (c) "Practical Software Measurement, Objective Information for Decision Makers", pages 168 – 171.

[11] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

2.5 QUALITY

2.5.1 Purpose

Tracking problem reports and overall software complexity provides insight to the quality of the developed product in terms of the design, documentation, the software itself, and other factors that effect cost, schedule, and system performance. This also provides insight into the efficiency of the defect identification and correction process(s).

Keep in mind that defects are not restricted to defects in the code only. Erroneous or poorly written requirements and/or design can have severe impacts on the quality of the code and are some of the most difficult problems to detect and correct using only formal testing. As a rule, a single software-testing phase will detect approximately 30% of defects. Against design errors, this can fall to as low as 10% and even lower for requirements errors. Additionally, if defects are not tracked until formal testing has occurred, errors that might have been detected during requirements analysis, design and coding phases, and corrected at a much lower cost, have now propagated into the formal software integration and systems integration testing where they are difficult and expensive to detect and correct.

Formal Peer Reviews of requirements, design and code can achieve defect detection rates of 60% to 75% per review. Measurements and metrics collected on peer review defect detection also serves to indicate the quality of the program much earlier in the life cycle when they can be corrected much more cheaply. Use of formal Peer Reviews will save approximately three hours of down stream testing and defect correction for every hour spent performing peer reviews. The return on investment for formal peer reviews is approximately 15 to 1[12]. Thus peer reviews not only help identify and correct quality problems early in the effort, but also significantly reduce the amount of time spent in formal testing.

Many developing organizations dislike reporting peer review defect results since it is believed that the large number of defects often found in these reviews give a false view of low quality for the program, especially to individuals unfamiliar with the process. If this is the case, the developing organization must provide some alternative method by which the quality of the development can be judged in the early phases of development, requirements analysis, design, code and unit test, when effective corrective action can be taken. Failure to do so will mean that if the program does have quality problems, they will only be correctable by means of a lengthy and expensive testing and rework phase.

2.5.2 Quality Attributes

The quality metric tracks the total number of problem reports by severity (per reference (b) IEEE/EIA 12207.2 pages 94-96 severity definitions), the number of closed problem reports, the problem reports which were opened during the reporting period, the age of the problem reports, and the complexity of the software under development. Problem reports are collected separately for design, documentation, and software problems. Other problems are those which do not fall into one of the three previous categories but adversely impact program performance, cost, or schedule.

Tracking of defects by CSCIs, builds, spirals and/or releases also helps to spot areas of the development which are experiencing more significant quality problems. If defect data is only viewed for

[12] Estimating Software Costs, T. Capers Jones, McGraw Hill, 1998, pp198, 199, 426, 478-480, 512-515.

the entire project, overall trends may appear to be acceptable. However there could be individual CSCIs with severe quality issues that will not be apparent at this level of abstraction.

As with other metrics, the developing organization must estimate the number of defects likely to occur. Such predictions must be based upon historical data from previous developments and provides the basis for determining the amount of testing and rework, which will be required in order to complete the effort. This helps to identify quality issues when defect rates exceed the expected rate, which may indicate inadequate testing and rework resources are available to reduce the number of open defects to an acceptable levels.

2.5.3 Specification of Quality Requirements

The buyer must clearly and contractually identify the minimum acceptable quality level for completion of the effort. Each delivered defect represents either an inability of the system to perform a mission essential capability, some degradation in its ability to perform a mission essential capability, or an increase in technical cost or schedule risk to the project or the life cycle support of the system. ***Minimum acceptable quality must be defined based upon what level of degradation in the ability of the system to perform mission essential capabilities is acceptable and/or the impact on the development or lifecycle support of the system.*** The following are guidelines for determining minimum quality levels for a system:

1. No Priority 1 or 2 defects[13].
2. Specify a maximum acceptable number of priority 3 defects per 1000 lines of source code or a maximum number for the entire system. Components of the software dealing with safety (aircraft navigation, weapons control, etc.) or interoperability may have much more stringent quality requirements than other less critical components. For these safety and interoperability components, separate and more stringent quality requirements may be specified than that required for the remainder of the system. It should also be considered how these defects will impact the ability of the operator to perform the mission. Systems operating in high stress and combat environments should have much more stringent quality requirements, which avoid increasing the operator workload and degrading the operators ability to perform the mission, than systems operating in a more benign environment.[13]
3. Specify a maximum acceptable number of priority 4 & 5 defects per 1000 lines of source code or a maximum number for the entire system. These are essentially operator inconveniences and other affects, which do not impact the ability of the system to perform, mission essential capabilities. Keep in mind the effect of these defects on the operator's ability to perform the mission when identifying maximum acceptable defect rates.

> *Note: Each defect represents either an inability of the system to perform a mission essential capability, some level of degradation in that ability, or an increase in technical, cost or schedule risk to the project or the life cycle support of the*

[13] SECNAVINST 5000.2B, Part 3 Program Structure Paragraph 3.4.3.1 Navy Criteria for Certification, subparagraph 17, and 3.4.3.2 Marine Corp Criteria for Certification, subparagraph 17. http://neds.nebt.daps.mil/5000.htm

system. ***Minimum acceptable quality must be established based upon what the maximum acceptable level of degradation in the performance of mission essential capabilities is and/or its impact on the development or lifecycle support of the system.***

2.5.4 Data Collection

2.5.4.1 Source

The developing agency will provide:

- ✓ Actual 'open' and 'closed' problem report data for each build and/or CSCI.
- ✓ The senility, or age, of each problem (problem reports open for less than 2 months, between 2 and 4 months, and greater than 4 months).
- ✓ Complexity data.
- ✓ Estimated number of "open" and "closed" problem reports for that time in the development.

2.5.4.2 Frequency

The reporting period is monthly after Software Requirements Review (SRR).

2.5.5 Quality Metrics

Software problem reports must be analyzed using three different views - status, age, and priority. Unless all three are reviewed, a complete picture of the program will not be available.

2.5.5.1 Software Problem Reports - Status

2.5.5.1.1 Purpose

To provide insight into the quality of the software under development.

2.5.5.1.2 Description

The numbers of new, total, and open Software Problem Reports (SPR) are graphed over time.

The data for this metric comes from the developer and any associated testing activities (which can include Independent Verification & Validation (IV&V) contractors and/or the government testers) and are a simple graph of the SPRs over time.

2.5.5.1.3 Analysis

While the absolute number of SPRs found during the program will hopefully be relatively small, the key is the trend line of the open SPRs.

If the quantity of open SPRs is seen to be increasing, the resources that are brought to bear on these SPRs must also be increasing.

Additionally, each SPR should be treated individually. A team should be in place to accurately investigate each SPR - no two SPRs will require the same amount of work to fix.

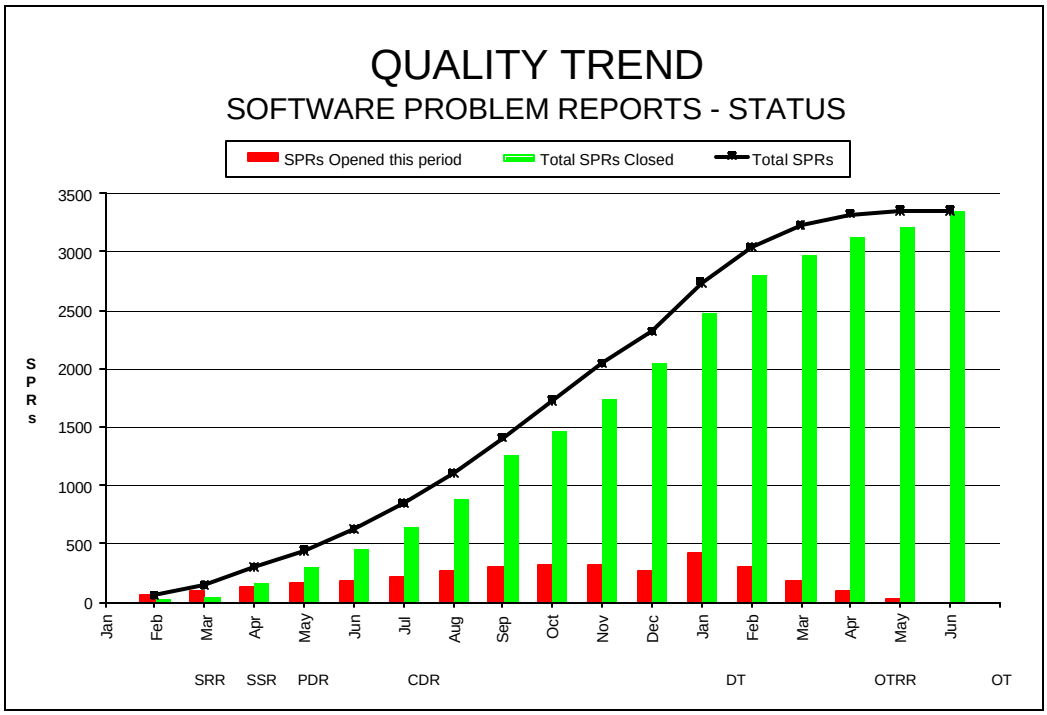
Consideration:	*If historical information is available from previous programs by the same developer, the numbers of SPRs for different points in the development can be estimated. Even without historical data, the "open" trend line can be monitored to assure sufficient effort is maintained on solving problems as they appear.*

2.5.5.1.4 Rule of Thumb

Increases in SPRs are frequently observed after major reviews (i.e., action items) and the start of testing activity (testing errors). If the increase is minor, it is necessary to investigate whether the product is of high quality or whether the review was ineffective. If there is a declining number of open SPRs, this may be caused by the reallocation of effort from finding problems to correcting problems (e.g., the testing effort is being applied to correction activities and not to testing). In this situation, test measurements and metrics should be reviewed to determine if testing is slowing down, falling behind schedule or if staffing in the test area has been reduced below planned levels.

2.5.5.1.5 Software Problem Report Status Chart Analysis Example

Figure 2-7 shows two sample charts. The first chart shows a program progressing normally. The second chart shows a program where problem reports remain open and a large number of problem reports have been opened during developmental testing.



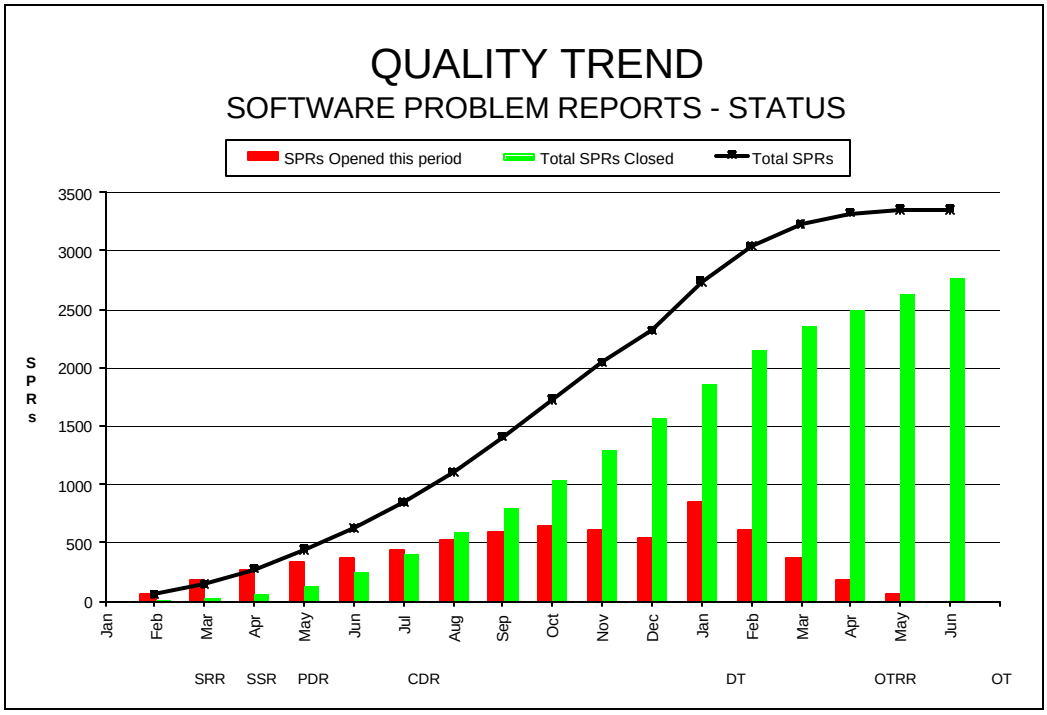


FIGURE 2-7. Software Problem Report Status.

2.5.5.2 Software Problem Reports - Age

2.5.5.2.1 Purpose

The Software Problem Report - Age metric provides information about the amount of time any given SPR remains open and therefore, unsolved.

2.5.5.2.2 Description

The numbers of open SPRs are graphed over time. Generally, three different age categories can be used, representing roughly 5% of the program time frame, e.g., for a 3-year program, categories of SPRs open for less than 2 months, between 2 and 4 months, and greater than 4 months can be used. A separate graph is made for each SPR priority.

2.5.5.2.3 Analysis

If SPRs are allowed to remain open, it may be a sign that difficult problems are being left outstanding without consideration. Generally it is exactly these SPRs that can drastically affect the design.

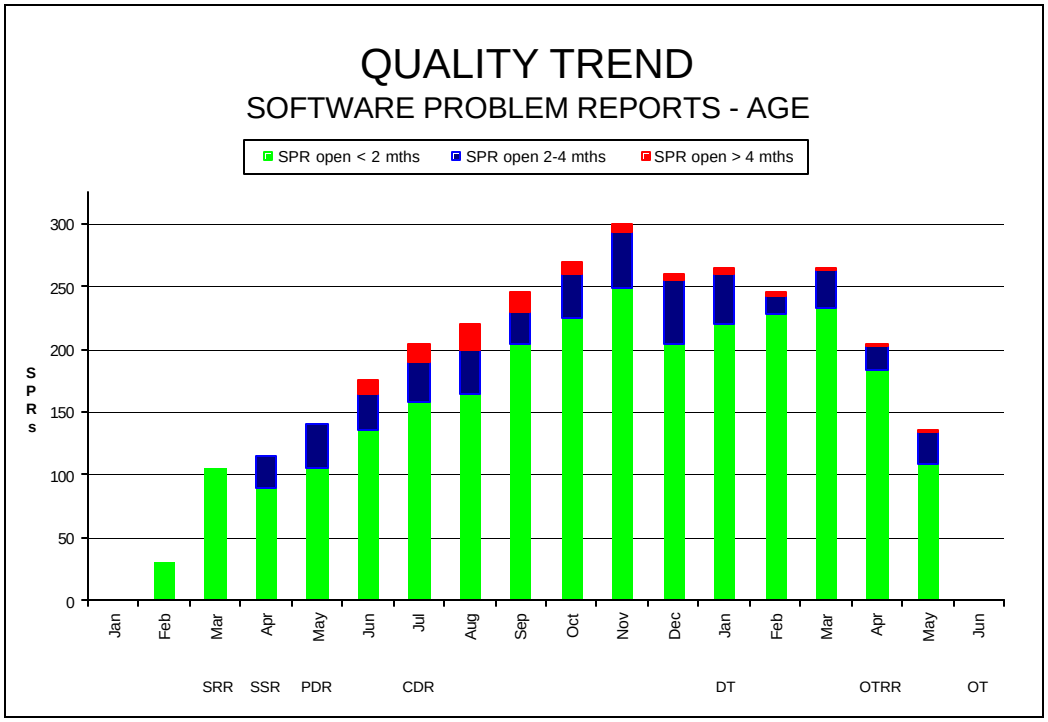
Therefore, not just the SPR closure rate, but also the age of the SPRs must be closely monitored.

2.5.5.2.4 Software Problem Reports Age Chart Example

Figure 2-8 shows two sample charts. The first chart shows a program progressing normally. The second chart shows a program with many problem reports still open late in the program.

> *Note: Due to the magnitude of the data, the SPRs scale on each chart is different. A reviewer must be aware of similar types of charts with differing scales.*

The SPRs must be carefully reviewed to assure that an abundance of "hard" problems or high priority problems are not left outstanding. A team should be in place to carefully review each SPR and ensure the proper resources are brought to bear to solve the problem promptly.



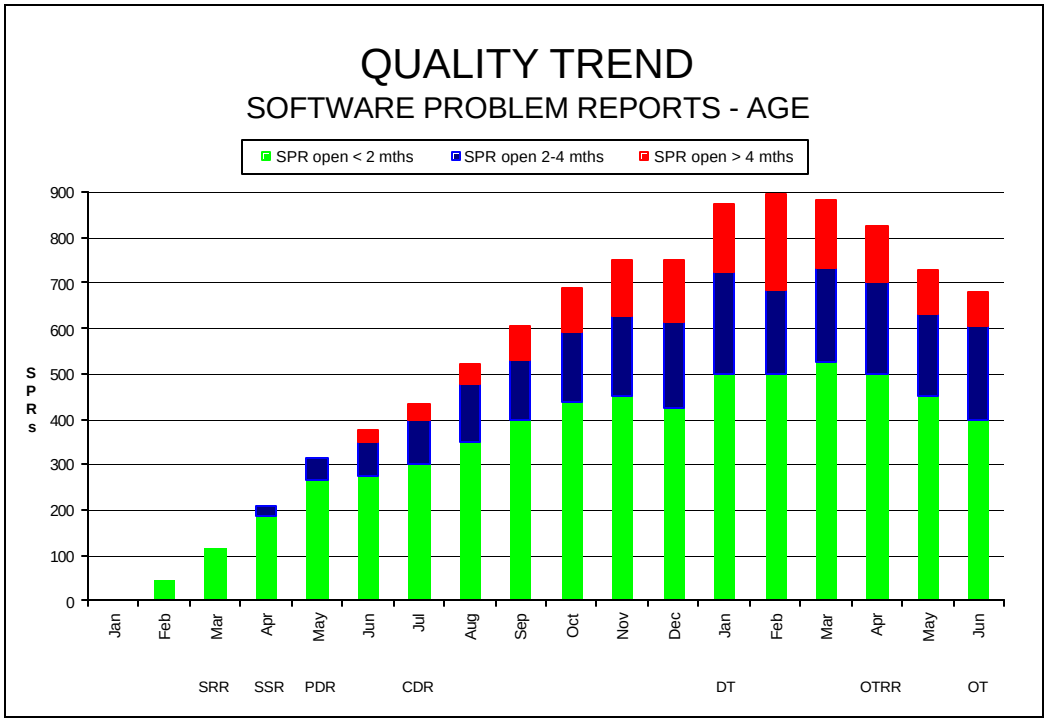


FIGURE 2-8 Software Problem Report Age.

2.5.5.3 Software Problem Reports - Priority

2.5.5.3.1 Purpose

This metric provides information about the priority or importance of open SPRs.

2.5.5.3.2 Description

The priorities of open SPRs are graphed over time. Generally, three different priority categories can be used representing have-to-solve versus nice-to-solve problems.

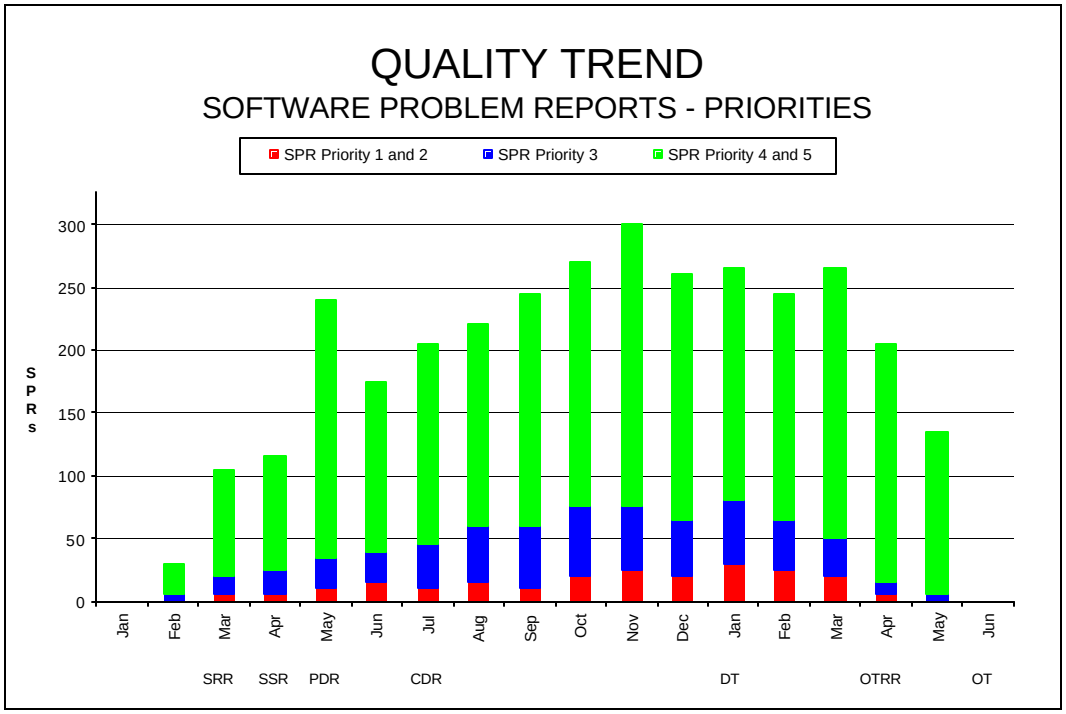
2.5.5.3.3 Analysis

Higher priority SPRs indicate defects which have a more significant impact on the ability of the system to perform its mission. The more and higher the priority of these defects the more significant the quality problems being experienced by the program.

2.5.5.3.4 Software Problem Reports Priority Chart Analysis Example

Figure 2-9 shows two sample charts. The first chart shows a program progressing normally. The second chart shows a program with many priority 1 and 2 problem reports still open late in the program.

Note: Due to the magnitude of the data, the SPRs scale on each chart is different. A reviewer must be aware of similar types of charts with differing scales.



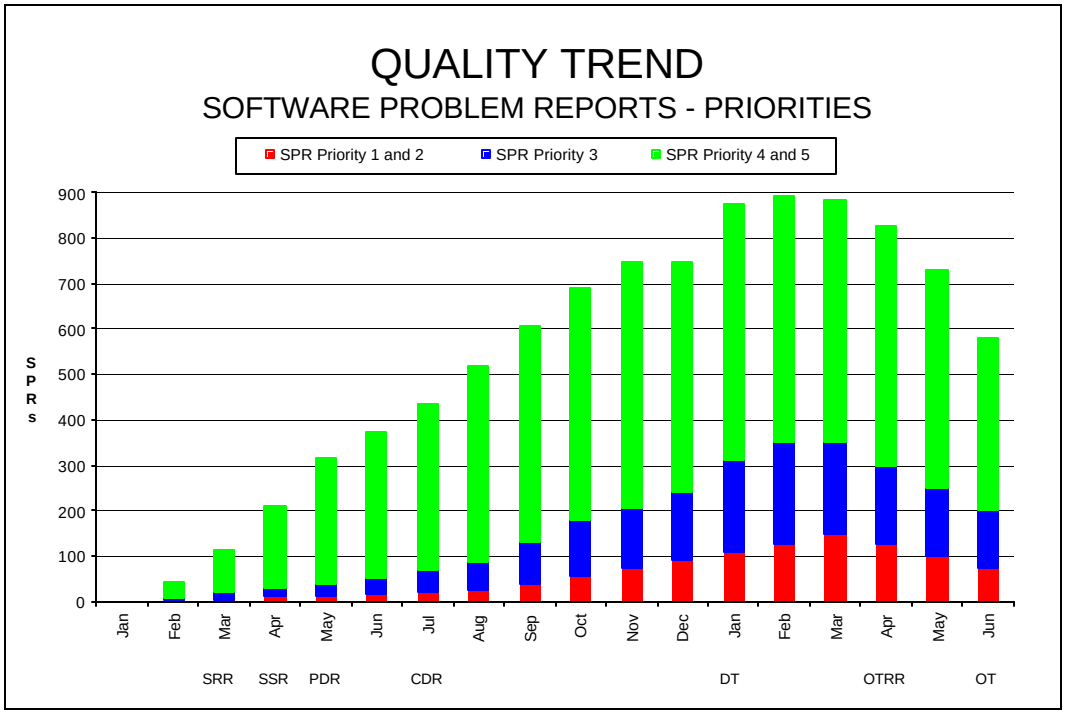


FIGURE 2-9. Software Problem Report Priorities.

2.5.5.4 Software Problem Reports – Predicted Versus Actual

2.5.5.4.1 Purpose

This metric provides information about the actual number of SPRs being generated versus the predicted number. This metric serves to identify quality problems and to indicate when the software has achieved the desired quality level.

2.5.5.4.2 Description

The predicted numbers of SPRs are based upon the number of SPRs injected during development of similar systems and the amount of time necessary to reduce the number to an acceptable level. The developer's schedule will be at least partially driven by the amount of time necessary to reduce the predicted number of SPRs to the quality level required by the buyer. The predicted number of open SPRs must reach or drop below the maximum number of SPRs acceptable to the buyer by the end of the development.

2.5.5.4.3 Analysis

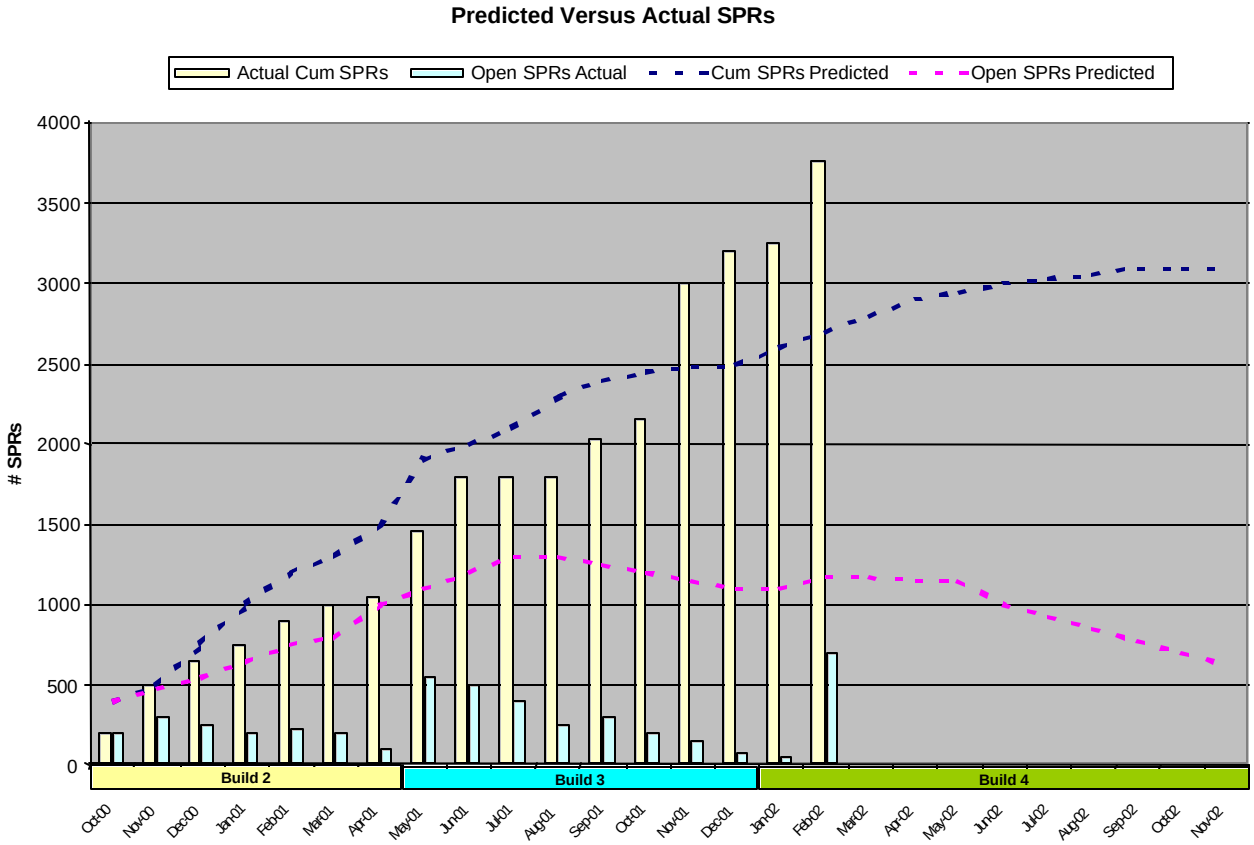
If the actual number of cumulative and/or open SPRs stay at or below the predicted numbers of cumulative or open of SPRs over the program schedule, this indicates the program is meeting its pre-deployment quality requirements. If the actual number of cumulative and/or open SPRs exceeds the predicted levels, this indicates that the quality is less than expected for the program. It further indicates that the resources allocated for rework may be inadequate since they were based on the number of defects being equal to or less than the predicted levels.

2.5.5.4.4 Software Problem Reports - Predicted versus Actual Chart Analysis Example

Figure 2-10 shows a sample of the measurement. The following should be noticed and considered:

1. Total number of SPRs stays below the predicted level until Nov 01 when it climbs steeply above the prediction line. At the same time, the number of "Open" SPRs is staying well below the Open SPRs predicted line. Thus, even though more SPRs than predicted are being generated, the developer is staying ahead of them. However, while the developer is handling the problem, the cause of the increase should be investigated.
    a. Is there some aspect of build 3 that is contributing to the quality problem?
    b. Were shortcuts taken during the build 3 development that could have resulted in an increase in the number of defects?
    c. How is the developer staying ahead of the increase in defects? Has additional staff been allocated to the task? If yes, what effect is this having on the program costs?
2. Notice how as each new build starts its test cycle, there is a jump in the number of "Open" SPRs. This is to be expected as testing of the new software and functionality in a build is initiated. Over time the number of "Open" SPRs continues to drop as testing continues which is what we want to see.
3. This particular metric does not break down the measurements by priority. Thus other measurements must be reviewed to ensure any SPR priority specific quality requirements are being

met. Even though the number of "Open" SPRs is well below the predicted level, if a large



percentage of these were priority 1 or 2 there would still be cause for concern.

FIGURE 2-10. Predicted Versus Actual SPRs

2.5.5.5 McCabe's Cyclomatic Complexity

2.5.5.5.1 Purpose

This metric allows the tracking of complexity by Computer Software Units (CSU).

2.5.5.5.2 Description

McCabe's Cyclomatic Complexity is a metric[14], which has a direct correlation to maintainability and reliability (high complexity leads to software that is difficult to maintain and is unreliable). The metric is calculated as follows:

$$v(G) = E - N + P$$

where:

[14] SEI Cyclomatic Complexity http://www.sei.cmu.edu/str/descriptions/cyclomatic.html

P = number of connected components

E = number of edges (transfers of control)

N = number of nodes (sequential group of statements containing only one transfer of control)

2.5.5.5.3 Analysis

The higher the complexity of a unit of code, the more difficult it is for the developer to understand what the unit of code is doing. This results in higher defect rates, higher life cycle maintenance costs, and eventually the code becomes essentially unmaintainable since it is so complex that any attempt to correct defects in the unit of code either fails or results in additional defects. SEI identifies the following level of risk with different levels of cyclomatic complexity.

- ✓ Cyclomatic Complexity of 1 – 10, a simple software module without much risk.
- ✓ Cyclomatic Complexity of 11 – 20, a moderately complex software module with moderate risk.
- ✓ Cyclomatic Complexity of 21 – 50, a complex, high-risk software module.
- ✓ Cyclomatic Complexity above 50, untestable software modules, very high risk. Such modules will be virtually impossible for developers to fully understand and difficult or impossible to develop effective test procedures for the modules. The complexity of such code will make the performance of full path testing even during unit testing extremely difficult or impossible.

Every attempt should be made to keep cyclomatic complexity for individual software modules at 10 or less. For modules above 10, if possible, they should be split into smaller modules. Due to the high risk associated with modules with a complexity above 20, such modules must be redesigned into smaller less complex modules.

2.5.5.5.4 McCabe's Cyclomatic Complexity Chart Analysis Example

Figure 2-11 shows a sample chart of the number of CSUs in different risk categories of Cyclomatic Complexity. As development progresses for the CSCI the percentage of the high risk CSUs is reduced throughout the coding and unit test phase. CSUs with high Cyclomatic Complexity will tend to be very high cost due to the high percentage of defects generated, and the difficulty in adequately testing and detecting such defects. Reducing the number of high Cyclomatic Complexity modules will tend to increase overall quality by decreasing defects.

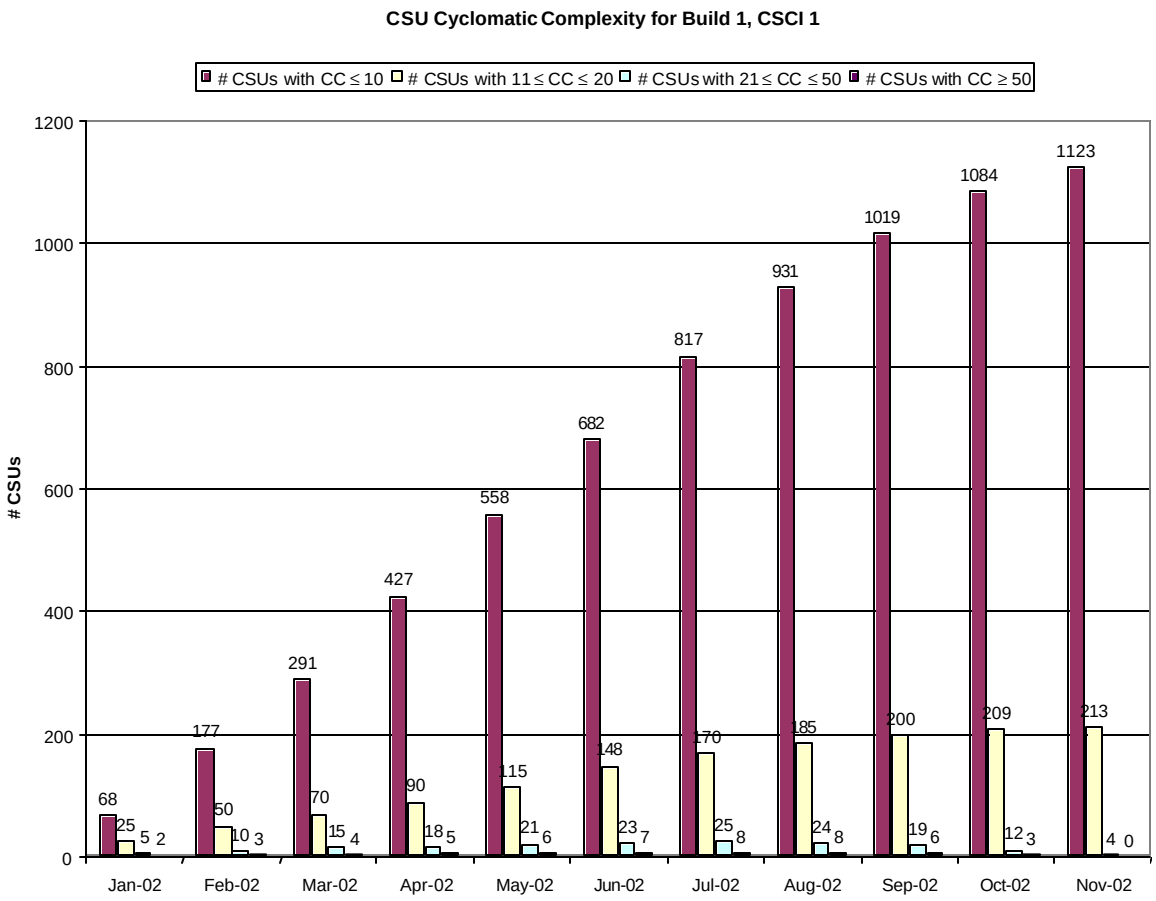


FIGURE 2-11. Computer Software Unit Cyclomatic Complexity

2.5.6 Maturity

2.5.6.1.1 Purpose

Evaluate the quality of a software intensive system to determine if it is ready for operational test.

2.5.6.1.2 Description

The maturity metric is the average number of Software Problem Reports (SPR) per test hour detected during a defined test period (weekly or monthly) depending on the length of the test phase. For example, if a one-month period is used and 320 hours of testing are performed, with 32 SPRs detected, then 0.1 SPRs per test hour have occurred. The SPRs per test hour should then be plotted over time. A decreasing maturity metric, which falls below a predetermined threshold, is an indicator that the software is ready for operational testing. The threshold is based on the success of previous software releases for the same or similar systems with a comparable level of complexity and testability. Even if the metric falls below the threshold, if there are open Priority 1 or 2 STRs, or all of the requirements of the software have not yet been tested, the software is not ready for operational testing.

2.5.6.1.3 Analysis

During the beginning of a test phase it is not unexpected to see an initial jump in the number of SPRs per test hour. This may be especially pronounced for the initial builds of a software development where there is a higher percentage of new, untested functionality in the system. As testing progresses, the SPR rate should drop as more SPRs are detected and corrected. In the final build of the development, when all requirements have been implemented, the SPR rate should be below the established threshold with a continuing downward trend before determining that the software is ready for operational testing. Reducing the SPR detection rate below the maximum acceptable level is only one of the criteria. Others that should be considered are:

1. Are there any priority 1 or 2 STRs? If yes the software is not ready for release.
2. Have all of the requirements been tested? If not the software is not ready for release.
3. Is the total number of open STRs less then that required for the system? If not, the software is not ready for release.

2.5.6.1.4 Maturity Chart Analysis Example

Figure 2-12 shows an example for a software development effort. SPRs per test hour are graphed for the systems integration and flight-testing phases for each of four builds. "Required SPR Detection Rate" delineates the threshold at the completion of Build 4, when full functionality has been implemented. "Linear Historical Maturity" shows the linear regression of historical SPR detection rates from previous developments. The "Linear Current Development" shows the results of a linear regression of the SPRs per test hour rate for the current development. Notice that the SPR detection rate per test hour rises significantly at the beginning of the Build 3 test cycle. This is expected given the large amount of new functionality that was implemented in Build 3. The overall trend for the number of SPRs detected per hour is decreasing and is well below the linearized historical rate. Throughout Build 4 testing the rate is below the threshold and is continuing a downward trend.

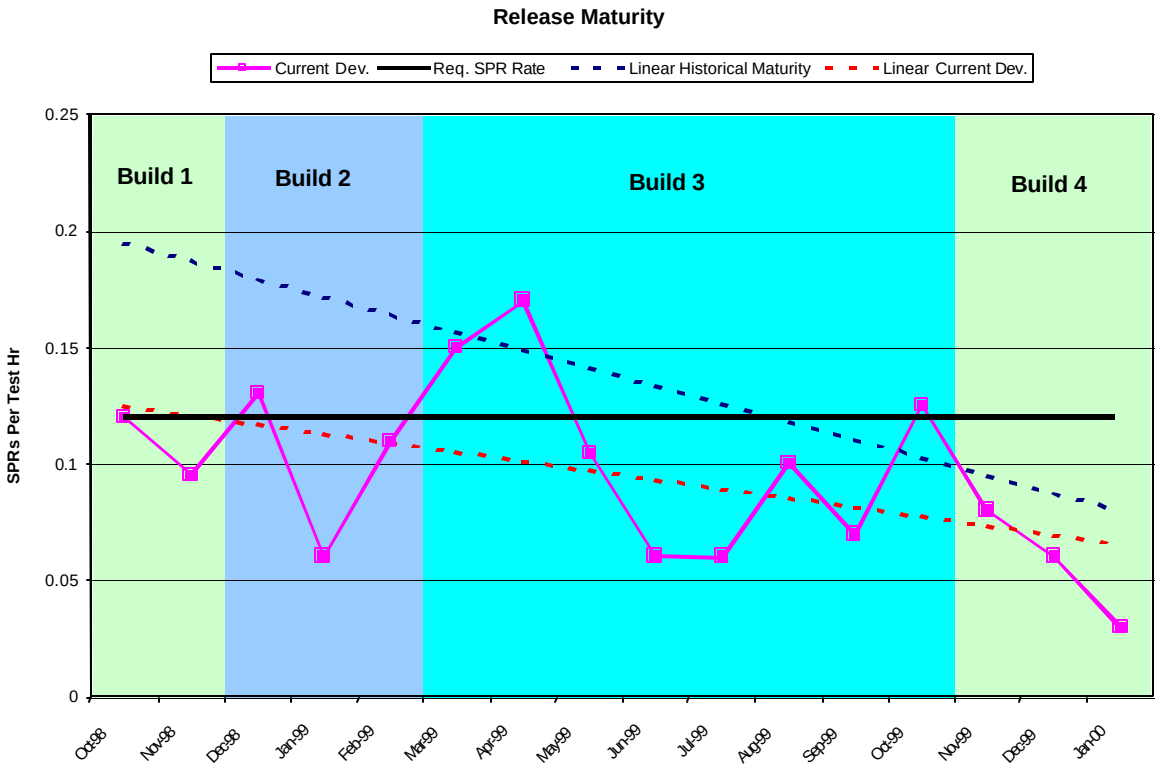


FIGURE 2-12. Software Release Maturity Metric

2.5.7 Quality & EVM Issues

As was discussed previously, defects result in the inability to perform an essential capability or various levels of degradation in the ability to perform the essential capability. This translates into system requirements, which are not implemented correctly. When the Government buys software, it wants the requirements for the software to be implemented. Thus a preferable means of employing EVM is to tie it to the implementation of requirements[15]. In each phase of software development, any outstanding defects should be equated to requirements incorrectly implemented and EVM appropriately reduced. For example:

- ✓ Assume at the end of the requirements analysis phase for a CSCI there are 1000 software requirements identified in the CSCI's Software Requirements Description (SRD). However due to formal peer reviews of the requirements or other document reviews, 50 of these requirements have defects identified against them. Only 95% of the earned value for the requirements analysis phase would be taken. If the planned Budgeted Cost of Work Scheduled (BCWS) for the CSCI's requirements analysis was $1M, and each software requirement is assumed to take the same amount of effort to develop. The Budgeted Cost of Work Performed (BCWP) at this point would be $950K. Schedule Variance (SV) would thus be -$50K. If the Actual Cost of Work Performed (ACWP) were $1.1M to reach this point, the Cost Variance (CV) would be -$150K.

[15] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

- ✓ Assume at the end of the design phase for the same CSCI there were, as a result of formal design reviews and other document reviews 40 requirements identified that were not correctly designed. In this case only 96% of the earned value could be taken at this point. If the BCWS for the CSCI's design phase was $2M, than the BCWP would be $1.92M with an SV of -$80K. If the ACWP was $2.1M to reach this point, CV would be -$180K.
- ✓ Assume at the end of the code and unit test phase for the same CSCI, there were as a result of formal code reviews and tests, 60 requirements that were not correctly coded. In this case only 94% of the earned value could be taken at this point. If the planned BCWS for the CSCIs code and unit test phase was $2M, than the BCWP would be $1.88M with a SV of -$120K. If the ACWP was $2.2M to reach this point, CV would be -$320K.
- ✓ Similar examples apply for other phases. In each case it was assumed that the developer planned to have everything done at the end of the phase. This is unrealistic since there are always defects preventing full implementation of all requirements at the end of any phase. A realistic plan would not assume or plan for zero defects at the end of each phase and would include a rework period to correct a predicted number of defects at the end of the phase.

Cyclomatic Complexity cannot easily be translated into a basis for taking earned value. Its purpose is to attempt to predict which module will experience quality problems due to an excessively complex design. The developer may choose, on the basis of historical data, to assume that some percentage of the software modules or CSUs will necessarily have a complex design, and allocate a higher than average BCWS to those CSUs based on this historical predictions.

2.5.8 Quality References

Further information on Quality measurements & metrics can be found in reference (c) "Practical Software Measurement, Objective Information for Decision Makers", on pages 179 – 184.

2.6 CAPACITY

2.6.1 Purpose

Computer Resource Utilization (CRU) tracks planned and actual percentage utilization of target computer resources and provides insight to the availability of hardware resources as the design progresses. Computing resource elements are: (1) processor throughput for each individual processing element, (2) memory for each processing element, and (3) Input/Output (I/O) for each individual bus or internal data network. CRU for the host development system is also tracked, as these resources can have an impact on cost and schedule of the development effort.

2.6.2 Description

Processor throughput is defined as the number of instructions per second that the processing element is capable of performing using a given mix of instructions (which should closely match the intended operational code). When calculating percentage utilization, the operational code together with any overhead functions (interrupts, operating system, parallel processing etc.) must be evaluated for a worse case scenario.

Memory utilization must be given for each type of memory found in the system (i.e., no single value as a conglomerate of all memory within the system), and is defined as the ratio of memory used to the amount of memory available. Examples might include 54% memory utilization for the MIPS R4000 local memory, 26% memory utilization of the common memory on the mission computer card, and 43% memory utilization of the bulk memory.

I/O bus utilization is defined as the ratio of the amount of bus bandwidth used to the amount of bandwidth available for each bus in the system. The amount of bandwidth used must not only include actual data but also overhead data and preamble, post-amble information.

2.6.3 Data Collection

2.6.3.1 Source

The developing agency will provide computer resource utilization data for each build.

2.6.3.2 Frequency

The initial CRU estimates are updated at major milestones up to and including Test Readiness Review (TRR). Planned and actual CRU is reported monthly after System/Subsystem Design Review (SSDR).

2.6.4 Capacity Metrics

2.6.4.1 Computer Resource Utilization (CRU) Usage

2.6.4.1.1 Purpose

This metric ties together the hardware and software by presenting the amount of hardware (memory, bus bandwidth, throughput) that is required to operate the given software.

2.6.4.1.2 Description

The amounts of memory, bus bandwidth, and throughput for each memory type, bus, and engine are plotted over time.

> *Note: For large systems, the number of processors, buses, and memory systems can be quite large. Each one must be tracked individually and NOT aggregated. Any one of these components can become a choke-point for the entire system.*

The developer must provide data showing the amount of useable hardware resources, and the estimation of the percentage of utilization over time.

The developer should indicate the multiplying factors that were used to convert a line of high-level code to the memory. The government must assure that these factors are met as the development moves from design through Code & Unit Test (C&UT).

Additionally, bus throughput is a critical factor that should have a small-dedicated team in place to monitor and control each message on each bus.

The developer must also predict the performance for each individual release or build of the software. Initial builds or releases should have predicted utilization levels well below the utilization requirements for the system. As more functionality is added to each succeeding build or release, the level of utilization for that build or release will increase. This will allow the developer to compare actual utilization for each build or release to predicted levels and give early indications of the accuracy of the predictions. If the utilization of early builds and releases exceeds the predicted utilization, even if it is still below the utilization requirements of the entire system, this indicates the developer has underestimated the utilization of the system and must immediately begin an effort to determine how to meet the utilization requirements. The sooner such build actuals can be evaluated to determine how they match predicated utilization requirements the better. Early engineering builds should be used in order to generate actual data on utilization as soon as possible in order to validate predicted utilization.

2.6.4.1.3 Analysis

Compiler updates and changes can severely affect the Computer Resource Utilization (CRU). If a compiler version change is introduced, the effect is generally positive (i.e., less code will be generated, requiring less memory and less throughput), but this is not always true.

Clock speed changes, together with a faster bus or better access to memory, may also increase the usable CRU.

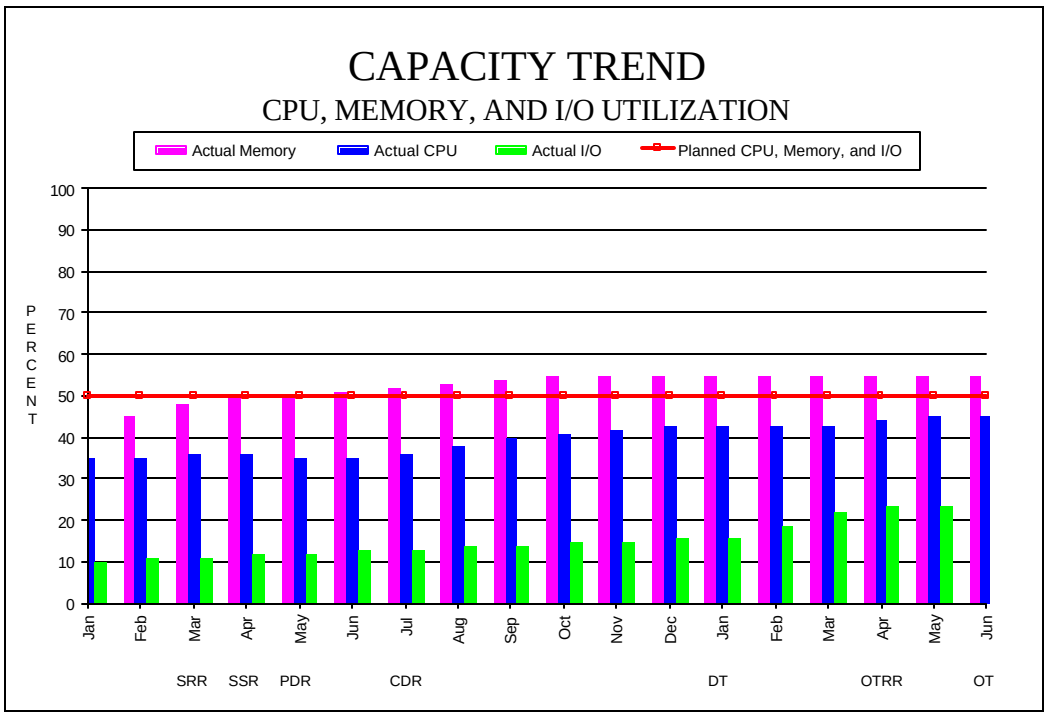
In any event, changes in requirements mean changes in code, which also ripples to the CRU.

2.6.4.1.4 CRU Utilization Chart Analysis Example

Figure 2-13 shows two sample charts. The first chart shows a program progressing normally. In this example it is assumed there is only a single processor, memory block and I/O bus so all may be shown on the same chart.

The second chart shows a program where reserve requirements have been used heavily. In its current condition, this system is not deployable to operational forces and may fail OPEVAL. The high percentage of memory and CPU utilization may have made it impossible for this system to meet its various performance requirement, especially the real time or near real time requirements. Additionally, there is virtually no room for enhancements and upgrades to the system, meaning the maintenance costs will be very high. The metrics have been clearly indicating there is such a problem since prior to September with no noticeable effort to take corrective action. Possible courses of action open to the program:

1. Delay OPEVAL while the code is completely redesigned to meet the utilization requirements. This will result in long schedule delays and large cost overruns and may be impossible without severely reducing the requirements and functionality of the system.
2. If it is a COTS based system it may be relatively easy to upgrade the system's CPU, memory and I/O in order to meet the original utilization requirements. While this will delay the program and increase the costs, it is likely to be much less expensive than trying to redesign the system.
3. A combination of the previous two options.



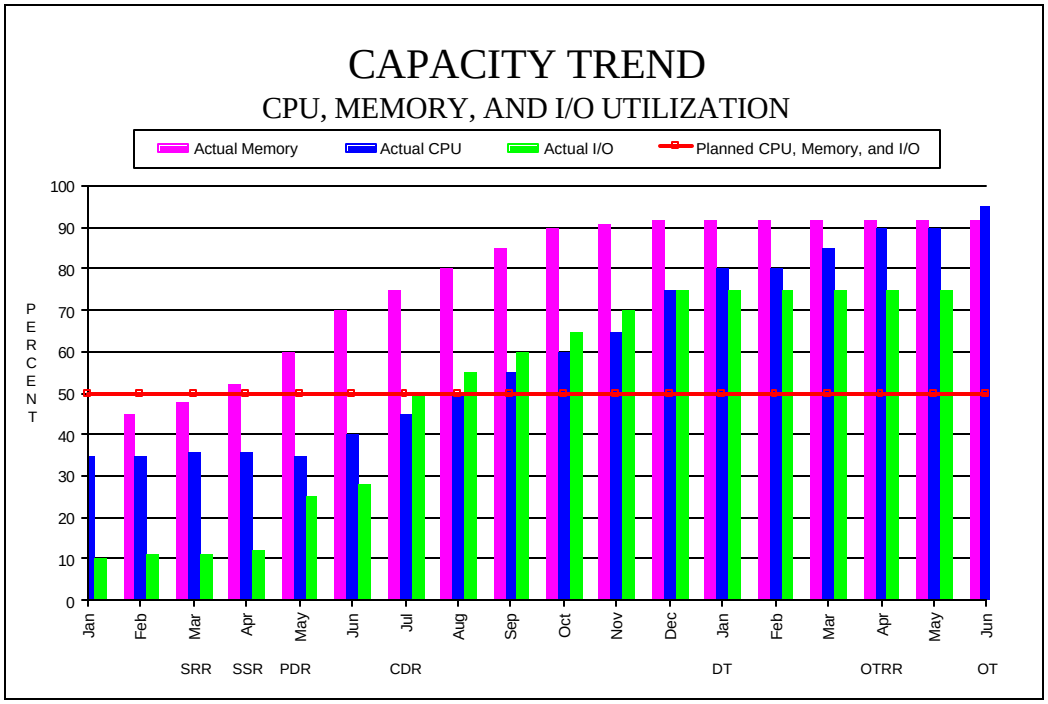


FIGURE 2-13. Computer Resources Utilization.

2.6.5 Capacity Utilization EVM Issues

Requirements for Capacity, similar to those relating to systems performance are often very difficult to track effectively using EVM[16]. A performance requirement for software usually requires the software to complete some tasks in a specified period of time, some examples of performance requirements are:

- ✓ Respond to an operator input within .5 second.
- ✓ Accept and process at least 1,000 radar contacts per second with no loss of data.
- ✓ Accept and process navigation data received at an 8-hertz rate with no loss of data.

Utilization and performance requirements can be much more difficult to implement since they can also encompass a range of other implementation requirements. For example:

- ✓ There is a requirement for processor A in the system to utilize no more than 50% of its processing capacity. Assume that processor A is using 75% of its processing capacity. This may require that the software executing on processor A be redesigned to make it more efficient in order to meet the 50% capacity utilization requirements. Assume that there are 100 requirements traced to the code running on processor A. Even though all these requirements may have been tested and found to be running correctly, because the utilization requirement is not being met, none of these requirements can be considered to be complete. These 100 requirements may all need to be extensively redesigned, coded and tested in order to meet the processor utilization requirement.
- ✓ Assume that the navigation system is required to accept 8-Hz navigation data and process it with no loss. Assume there are 50 other requirements dealing with how the navigation data will be processed. If the 8 Hz processing requirement is not being met, than none of the other 50 requirements can be considered met since they may have to be completely redesigned, re-coded and re-tested in order to meet the 8 Hz performance requirement.

From an EVM perspective, this means that if a performance or utilization requirement is not being met, all the other requirements describing how to implement the software which is related to that utilization or performance requirement cannot be considered to have been met either, or at least not completely met. Thus, if such an issue occurs, one option is not to take EVM for any of the related requirements which are associated with the implementation of the performance or utilization requirement. However, since problems with performance or utilization requirements may not be discovered until the system is in coding, software integration testing or later, this can be extremely messy. By this time the earned value has probably been taken for the design, coding and some of the testing of the software related to these implementation requirements. It may be only practical to not take earned value starting at the point where the failure of a performance or utilization requirement has been discovered. For example:

- ✓ Assume that the software for a system will be implemented in 4 builds. Assume that the system is required to utilize no more than 50% of bus A when all the requirements of the system are implemented. As part of the development plan for build 1, it is determined that no more than 25%

[16] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

of the bus A throughput should be utilized when all the requirements of build 1 are implemented. During Systems Integration testing for the first build, it is discovered that the IO utilization on bus A is actually 35%. Assume that it is also determined that there are 100 implementation requirements in build 1 that require the use of bus A. One option for earned value might be to not take any further Earned Value for these 100 requirements until the bus A utilization requirement problem was resolved. If there were 1000 requirements in build 1, and the BCWS was $10M for build 1 systems integration, than the BCWP would be $9M and the SV -$1M. If the ACWP was $11M to conduct build 1 systems integration testing, this would result in a CV of $2M.

- ✓ It might be concluded for the previous case, that some credit should be given for the correctly implemented requirements, even though the bus A utilization requirement had not been met. It could be decided that; 50%, 25%, or some other amount of the earned value for the 100 requirements would be withheld until the bus A utilization requirement had been met. This would depend on the systems and software engineering's analysis of how much rework would be required of the existing software in order to meet the utilization requirement. If 25% were withheld, than the SV would be $250K and the CV would be $1,250K.
- ✓ For subsequent builds of the system, the same withholding of earned value would also occur until the bus A utilization requirement had been corrected. Thus assume that in build 2 an additional 100 requirements would be implemented which would utilize bus A. Assume the entire cost for build 2, BCWS, was $30M and the cost for all phases of the development of these 100 bus A requirements in build 2 was $3M. If the bus A utilization problem was not corrected by the end of build 2 and the same 25% withhold was used, this would mean the BCWP was $29,250K, if everything else for build 2 was implemented correctly. The SV would therefore be -$750K. If the ACWP for build 2 were $32M, the CV would be -$2,750K.
- ✓ Keep in mind that there are a variety of ways the bus A utilization requirement might be resolved. The code might be redesigned to meet the utilization requirement. The Government may change the requirement to make it easier to meet. Bus A might be upgraded to a higher bandwidth bus. A combination of all three may be used to solve the problem.

The EVM methods described above result in a very heavy penalty for failure to meet a utilization or performance requirement since it results in reducing or eliminating earned value for any other requirement that relies on the resource being utilized or contributes to meeting the performance requirement. This may seem unreasonable, until one considers the cost and schedule delay that could impact the program if it is necessary to conduct a radical redesign of the software architecture in order to meet the utilization and or performance requirement. Such problems are notorious for being extremely expensive to resolve in software. It is often much cheaper to resolve these problems with upgrades to the CPU, memory, IO or whatever other resource is bottlenecking the software. Failure to adequately weight such a problem will result in an unrealistically high CV and SV, which will not reflect the difficulty of correcting such a problem.

2.6.6 Capacity Utilization References

Additional information on measurements and metrics for capacity utilization can be found in reference (c) "Practical Software Measurement, Objective Information for Decision Makers", pages 184 – 186.

2.7 SCHEDULE

2.7.1 Purpose

The schedule performance measure tracks the planned date and the actual date for each milestone and provides an understanding of how all major events for the program development effort are related. Attainment or non-attainment of program milestones can be an indicator of general program well being. CSU schedule performance can be tracked during the design, code, and test phase. Software builds can be tracked as appropriate.

2.7.2 Description

See Appendix C for a list of IEEE/EIA 12207 reviews and milestones along with their corresponding MIL-STD-498 and DoD-STD-2167A reviews and program milestones. It is imperative that entry and exit criteria are well defined for each milestone and that the developer and government agree to these definitions.

It is highly recommended that original schedules be kept and compared to current actual and projected schedules. Original schedules should not be changed unless the Program Manager is fully aware of why the schedule change is necessary. Floating baseline schedules cause programs to appear always on schedule, always compare the current schedule with the last one received.

2.7.3 Data Collection

2.7.3.1 Source

The developing agency will provide milestone performance data.

2.7.3.2 Frequency

The milestone performance estimates are updated monthly after contract award.

2.7.3.3 Format(s)

Ideally program schedules should be provided in Microsoft Project or some other project-planning tool by the developer. Both planned and actual completion dates for the various tasks and Work Breakdown Structure (WBS) items in the schedule must be identified.

2.7.4 Schedule Metrics

Developing agencies' performance may be tracked by monitoring CSU (Computer Software Units) Design, Code, and Test performance, and Major Milestone achievement. A rule of thumb is that late or unacceptable software schedules are often good indicators of schedule risk and bad software products.

2.7.4.1 CSU Design, Code, and Testing Tracking

2.7.4.1.1 Purpose

Provides insight into the rate at which CSUs are completing design, coding and testing and how it corresponds to the project plan and schedule.

2.7.4.1.2 Description

CSUs are low level software routines and functions. In Object Oriented systems CSUs roughly correspond to methods, functions and other executables in the assorted classes. In an Object Oriented development it may be determined that tracking the development of classes is more effective than CSUs. In this discussion CSUs will be utilized.

2.7.4.1.3 Analysis

Evaluation of the developing agency performance during software design, coding, and testing provides insight into the developer's ability to achieve planned schedules.

When evaluating progress of the CSU development, it is essential to determine if the planned functionality/requirements are being implemented. The progress in CSU development must be evaluated to determine if the software requirements allocated to those CSUs are actually being implemented. For example: assume that on 1 Jan 02, 300 CSUs are planned to have completed coding and have implemented 100 software requirements. However, while it turns out that 300 CSUs have completed coding on 1 Jan 02, only 75 software requirements have been completed. This indicates that the program is behind schedule. The number of CSUs necessary to implement the software requirements have been under estimated, or the size has been underestimated, or both.

2.7.4.1.4 CSU Design, Code, and Test Chart Analysis Example

Figure's 2-14 through 2-16 show development progress for CSCI A, build 1. Figure 2-13 shows actual progress against plan of the CSUs completing detailed design. Figure 2-14 shows actual progress against plan for CSUs completing Code & Unit Test (CUT). Figure 2-15 shows actual progress against plan for CSUs completing Integration Testing. In each of the figures, both planned and actuals of the numbers of software requirements planned to be detailed designed, CUT or integration tested as a result of the CSUs implementation are also identified. This allows a comparison to be made to ensure that the number of CSUs is sufficient to meet the planned functionality. Notice that for detailed design and CUT, the planned number of CSUs have been completed, although there is approximately a 3-month delay. However, the number of requirements implemented is significantly below the number planned, 103 versus 145. This indicates that the effort is even further behind than the three-month schedule slip in completing the planned 290 CSUs indicates. Combined with the schedule slip, it seems obvious at this point that the number of CSUs has been underestimated and not enough time has been allocated to the effort. Some other possible questions and considerations are:

- ✓ How does the planned staffing for the effort compare against the actual staffing for this effort? Is the problem completely or partially caused by inadequate staffing?
- ✓ If staffing is comparable with the planned levels, and if the size (SLOC, FPs, etc.) is the same as estimated, why is estimated productivity running so far behind actual productivity (SLOC/hr)? Is the staff less experienced or skilled than what was originally planned? Are the requirements more complex and difficult to implement than expected?
- ✓ What about the size (SLOC, FPs, etc.) of the CSUs? Has the size increased over what was originally predicted? If this were the case, it would account for the effort falling behind in the implementation of requirements even if the staffing and productivity was at predicted levels.

Notice that the total number of CSUs completing integration testing is less than the number having completed detailed design and CUT, 276 versus 290. The number of requirements that completed integration testing was also less than the number, which completed detailed design and CUT, 99 versus 103. This is most likely due to failed test procedures. Since the test procedure failed, the associated CSUs and requirements cannot be considered complete. Requirements, which were either not implemented or failed testing in this build, will have to be completed at some later date in the development of a future build. This will further increase the workload in these future builds causing further delays. Based on the performance achieved in build 1, the number of CSUs planned for future builds are probably inadequate for the planned for requirements and the amount of cost and schedule required is also probably inadequate. Future build schedules should be revised based on the build 1 actuals.

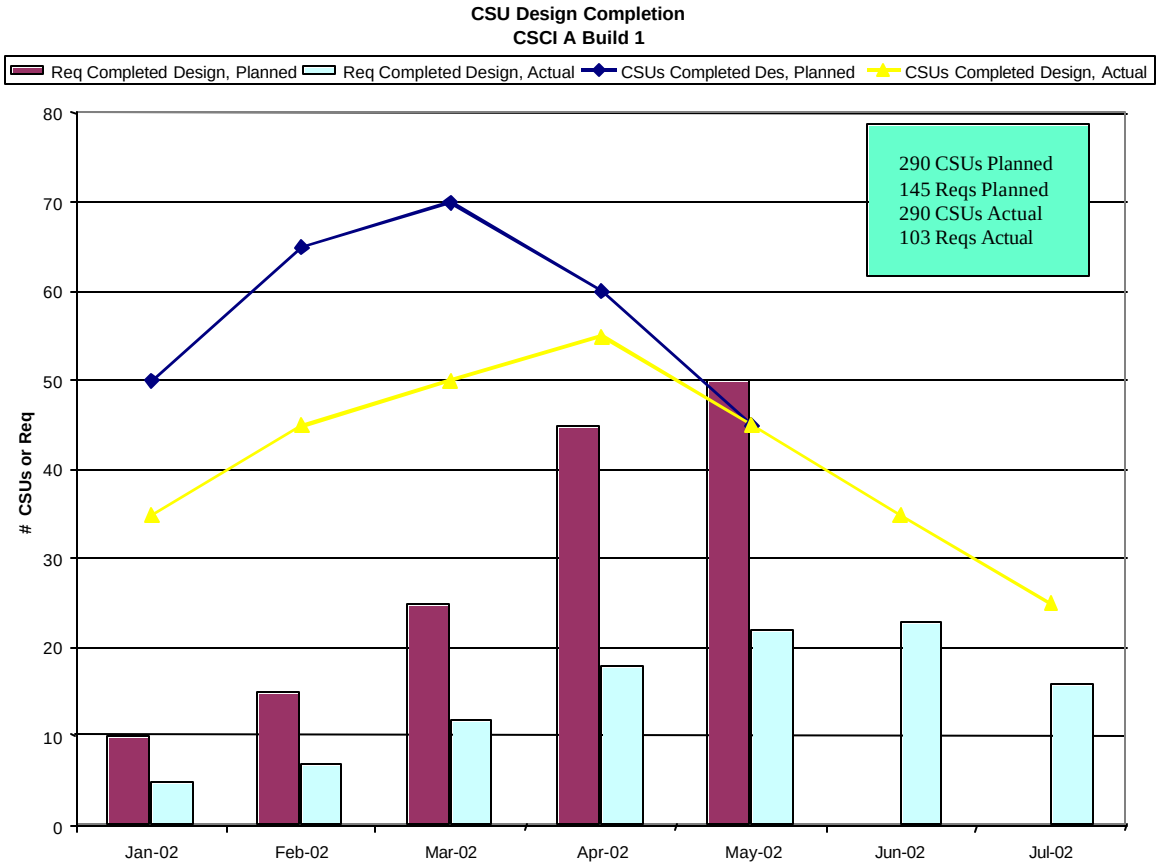


FIGURE 2-14. CSU Design Completion

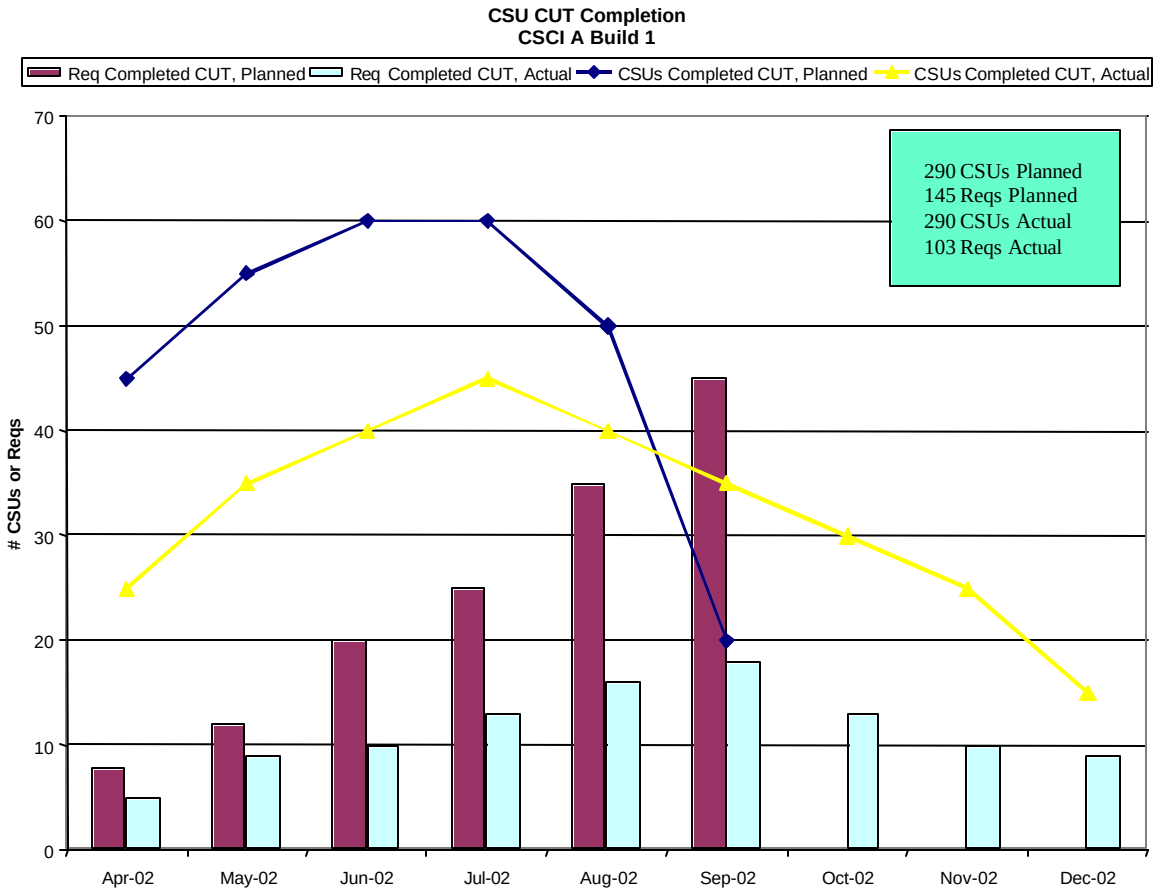


FIGURE 2-15. CSU Code & Unit Test (CUT) Completion

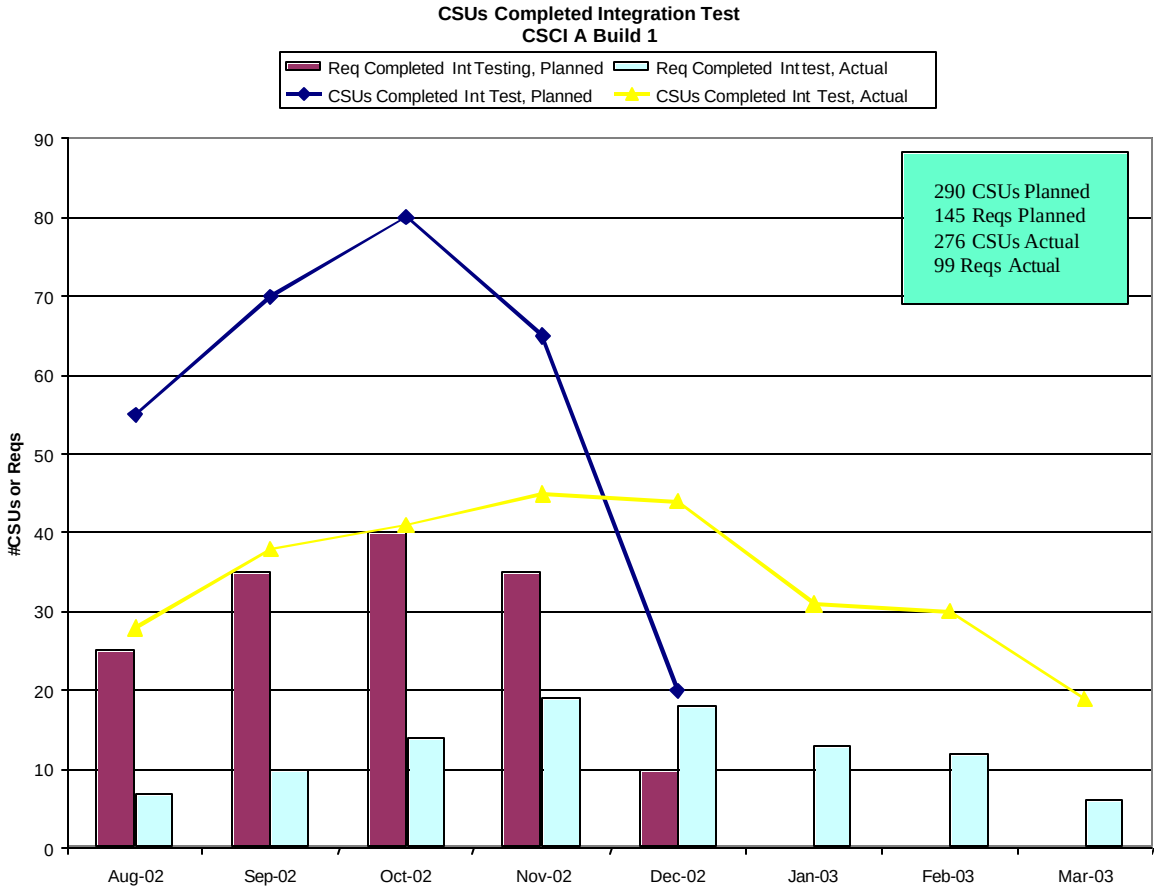


FIGURE 2-16. CSUs Completed Integration Test

2.7.4.2 Major Milestone Tracking

2.7.4.2.1 Purpose

Track progress of the program in achieving milestones.

2.7.4.2.2 Description

There will be various programmatic and technical reviews conducted during a software development. Each of these reviews must have specifically identified exit criteria, see NAVAIR INSTRUCTION 4355.19B Systems Engineering Technical Review Process[17]. Appendix C contains a list of IEEE/EIA 12207 reviews and milestones and the corresponding MIL-STD-498 and DOD-STD-2167A reviews and milestones.

2.7.4.2.3 Analysis

It is essential that all of the exit criteria be completed before a milestone can be considered to have been achieved. For example: assume that the first build of the development is planned to contain 1000 requirements. However, even though it is delivered on time and on cost, only 750 requirements have

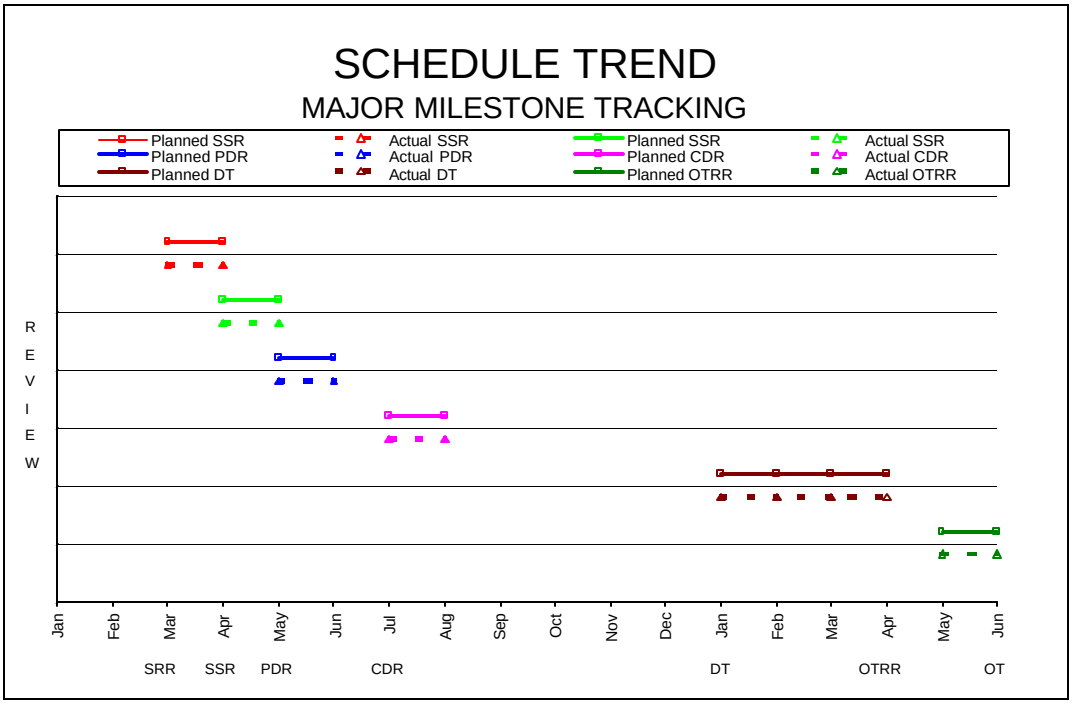
[17] NAVAIR INSTRUCTION 4355.19B Systems Engineering Technical Review Process
https://directives.navair.navy.mil/index.cfm

been implemented. The milestone has not been completely met. There are an additional 250 software requirements that must be squeezed into a later point in the development. Eventually it will no longer be possible to hide cost and schedule overruns by declaring milestones met even though all of the requirements have not been implemented. Another type of important milestone is the delivery dates for hardware and software not being developed as part of the effort. Late delivery of Government Furnished Equipment (GFE), COTS hardware and software, or hardware and software being delivered separately from the effort being tracked can have a major impact on the project.

Declaring technical and programmatic reviews complete when they have not met all of the exit criteria can also be very risky. Assume that the Software Requirements Review is declared completed on its scheduled date, even though a large percentage of requirements analysis is incomplete. This will result in the remaining requirements analysis being completed during design, code and unit test and possibly even later test phases. While some level of concurrency is desirable and expected between software development phases, too much concurrency results in low quality due to rework to correct design and code based on immature incomplete requirements. This results in higher costs, longer schedule and reduced functionality being delivered to the fleet.

2.7.4.2.4 Major Milestones Chart Example

Figure 2-16 shows two sample charts. The first chart shows a program progressing normally. The second chart shows a program behind schedule.



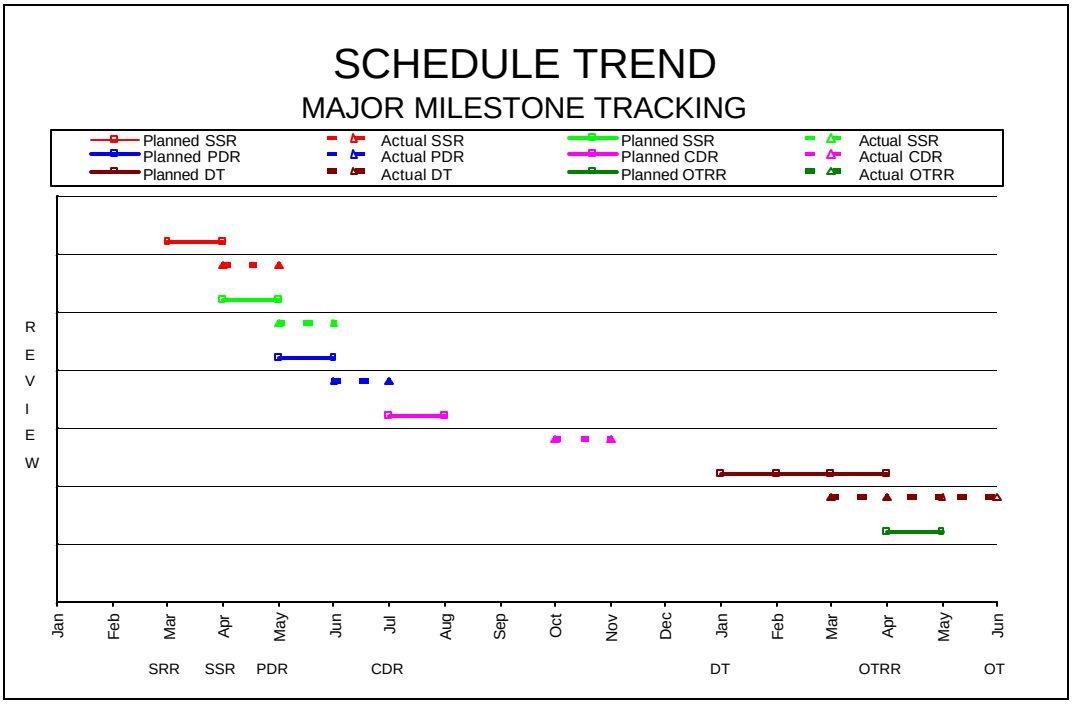


FIGURE 2-17. Overall Schedule.

2.7.5 Schedule EVM Issues

As has been previously discussed, the allocation of earned value should be closely tied to the implementation of software metrics. While the schedule metrics, as with all metrics provide valuable insight into the progress of the development, they can be misleading if they are not accurately related to the implementation of requirements.

In the discussion of CSU design, CUT and Integration Testing measurements and metrics, while the number of CSUs planned to be designed and coded was achieved, the number of software requirements was significantly lower than planned. If Earned Value had been based entirely on the number of CSUs actually completed in the design phase or the number of CSUs coded for the code & unit test phase the BCWP and thus the SV and CV would be too high[18]. While the planned number of CSUs had been designed and coded, the number of planned requirements had not been implemented. The actual number of requirements implemented must be considered when allocating EVM if an unrealistically high SV and CV is to be avoided.

2.7.6 Schedule References

Additional information on schedule measurements and metrics can be found in reference (c) "Practical Software Measurement, Objective Information for Decision Makers", pages 161 – 168.

[18] See Appendix D & http://www.acq.osd.mil/pm/ for further information on EVM implementation.

3. CONTRACT APPLICATION

3.1 CONTRACT & RFP WORDING REFERENCES

See the latest version of the NAVAIR Acquisition Guide[19] for suggested software measurement and metrics for Requests For Proposals (RFP), Statements Of Work (SOW), Contracts and Contract Data Requirements List (CDRLs).

Additional Information on suggested wording for RFPs, SOWs, Contracts and CDRLs can be found in reference (f). While reference (c) has replaced the remainder of reference (f), the Department of Defense Implementation Guide for reference (f) is the most current available.

[19] NAVAIR Acquisition Guide, AIR-1.1.1, http://www.nalda.navy.mil/acquisition/tools.html or http://www.deskbook.osd.mil/

APPENDIX A. Acronym and Abbreviations

A

ACWP	Actual Cost of Work Performed
AIR-4.1	Systems Engineering Department
AIR-4.1.11	Software Engineering Division

B

BCWS	Budgeted Cost of Work Scheduled
BCWP	Budgeted Cost of Work Performed

C

CUT	Code & Unit Test
C&UT	Code and Unit Test
CCP	Code Counting Program
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integrated
COTS	Commercial-Off-The-Shelf
CPI	Cost Performance Index
CRU	Computer Resource Utilization
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
CV	Cost Variance

D

DT	Developmental Test

E

EVM	Earned Value Management

F

(none)

G

GFE	Government-Furnished Equipment

H

(none)

I

I/O	Input/Output
IPT	Integrated Program Team
IV&V	Independent Verification and Validation

J

(none)

K

(none)

L

(none)

M

(none)

N

NAVAIR	Naval Air Systems Command

O

OPEVAL	Operational Evaluation
ORD	Operational Requirements Document

P

PDR	Preliminary Design Review

PSM	Practical Software Measurement
PSM	Practical Software & Systems Measurement

Q

(none)

R

RFP	Request for Proposal

S

SAM	Surface-to-Air Missile
SDP	Software Development Plan
SEI	Software Engineering Institute
SLOC	Source Lines of Code
SM&M	Software Measurement and Metrics
SOW	Statement of Work
SPI	Schedule Performance Index
SPR	Software Problem Report
SRD	Software Requirements Description
SRR	System Requirements Review
SRS	Software Requirements Specification
SSA	Software Support Activity
SSR	Software Specification Review
SSS	System/Subsystem Specification
SV	Schedule Variance

T

TECHEVAL	Technical Evaluation
TRR	Test Readiness Review

U

(none)

V

V&V	Verification and Validation

W

WBS	Work Breakdown Structure

X

(none)

Y

(none)

Z

(none)

APPENDIX B. Additional Reference Material

Boehm, B., Software Engineering Economics, Englewood Cliffs, NJ, Prentice-Hall 1981.

Florac, W., Software Quality Measurement: A Framework for Counting Problems and Defects, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-xx-TR-xx Mmm 199x.

Goethert, W., Software Effort and Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-xx-TR-xx Mmm 199x.

Humphrey, W., Managing the Software Process, Reading, MA, Addison-Wesley Publishing Co. 1989.

Institute of Electrical and Electronics Engineers, Standard for a Software Quality Metrics Methodology, *P-1061/D21*.

McAndrews, D., Establishing a Software Measurement Process, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-xx-TR-xx Mmm 199x.

Park, R., Software Size Measurement: A Framework for Counting Source Statements, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-92-TR-22 Sep 1992.

Rozum, J., *NAWCADWAR* -- Software Measurement Guide, Software Engineering Institute, Pittsburgh, PA, SEI/NAWC-92-SR-1 Oct 1992.

Rozum J., Software Measurement Concepts for Acquisition Program Managers, Software Engineering Institute, Pittsburgh, PA, CMU/SEI-92-TR-11 Jun 1992.

APPENDIX C. Comparison of Software Life Cycle Standards

Comparison of MIL-STD-498 development activities to 12207:

MIL-STD-498 Development Activities	**IEEE/EIA 12207.0 Development Activities**
5.1 Project planning and oversight 5.2 Establish software devel. environment	5.3.1 Process implementation
5.3 System requirements analysis	5.3.2 System requirements analysis
5.4 System design	5.3.3 System architectural design
5.5 Software requirements analysis	5.3.4 Software requirements analysis
5.6 Software design	5.3.5 Software architectural design 5.3.6 Software detailed design
5.7 Software implementation and unit testing	5.3.7 Software coding and testing
5.8 Unit integration and testing	5.3.8 Software integration
5.9 CSCI Qualification testing	5.3.9 Software qualification testing
5.10 CSCI/HWCI integration and testing	5.3.10 Software integration
5.11 System qualification testing	5.3.11 System qualification testing
5.12 Preparing for software use	5.3.12 Software installation
5.13 Preparing for software transition	5.3.13 Software acceptance support
5.14 Software configuration management	6.2 CM Process
5.15 Software product evaluation	6.7 Audit Process
5.16 Software quality assurance	6.3 QA Process
5.17 Corrective action	6.8 Problem resolution Process
5.18 Joint technical and management reviews	6.6 Joint review Process
5.19.1 Risk management	.2 Annex L - Risk Management
5.19.2 Software management indicators	.2 Annex H - Software measurement categories
5.19.3 Security and privacy	
5.19.4 Subcontractor management	6.3.3.3 Assure prime requirements passed to subs
5.19.5 Interface with software IV&V agents	
5.19.6 Coordination with associate developers	
5.19.7 Improvement of project processes	7.3 Improvement Process

Comparison of Reviews

DoD-STD-2167A/MIL-STD-1521B Formal Reviews	**MIL-STD-498: Joint Reviews**	**12207.0 Joint Review Process**
	Joint Technical Reviews	Technical reviews

	Joint Management Reviews	Project management reviews
	Software plan reviews	Software plan reviews
System Requirements Review (SRR)	Operational concept reviews, System/subsys reqts review	Operational concept reviews, System/subsys reqts review
System Design Review (SDR)	System/subsys design review	System/subsys design review
Software Specification Review (SSR)	Software requirements review	Software requirements review
Preliminary design Review (PDR)	Software design review	Software design review
Critical Design Review (CDR)		
Test Readiness Review (TRR)	Test readiness review Test results review Software usability review Software supportability review Critical requirements review	Test readiness review Test results review Software usability review Software supportability review Critical requirements review
Functional Configuration Audit (FCA)		
Physical Configuration Audit (PCA)		

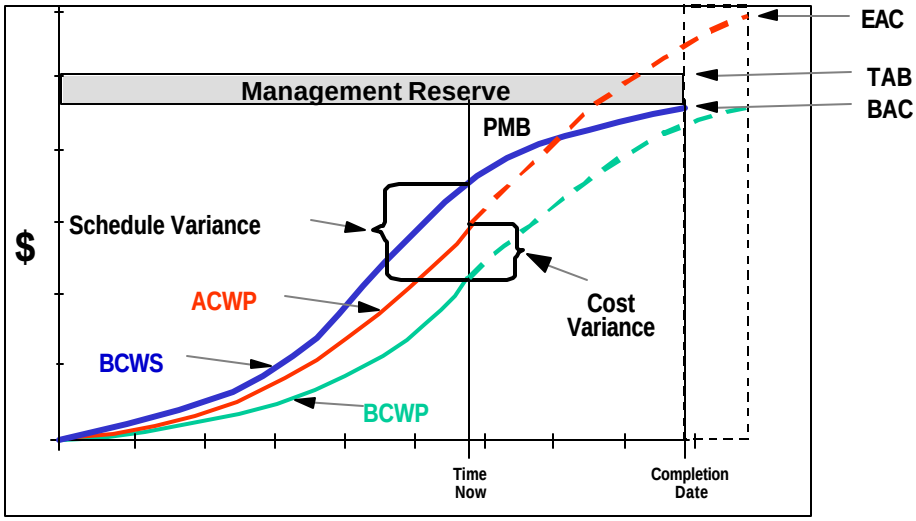
Comparison of documents among software standards

	MIL-STD-2167, MIL-STD-498, J-STD-016 Document	**MIL-STD-498 DI-IPSC-**	**J-STD-016**	**12207.0 Clause**	**12207.1 Information Item**
SDP	Software Development Plan (SDP)	81427	E.2.1	5.3.1.4 5.2.4	6.5 Development process plan 6.11 Project management plan
STP	Software Test Plan (STP)	81438	E.2.2	5.3.8	6.18 Software integration plan
IP	Software Installation Plan (SIP)	81428	E.2.3	5.5.5.2 5.3.12.1 7.4.1.1	Migration plan, Software installation plan, Training plan
CRISD	Software Transition Plan (STrP)	81429	E.2 4.	5.5.1.1 5.5.5.2 5.5.5.2 5.4	Maintenance plan, 6.8 Maintenance process plan, Migration plan 6.9 Operation process plan
SSDD	Operational Concept Description (OCD)	81430	F.2.1	5.1.1.1	6.3 Concept of operations description
SSS	System/Subsystem Spec (SSS)	81431	F.2.2	5.1.1.2 5.3.2	6.26 System requirements specification
IRS	Interface Requirements Spec (IRS)	81434	F.2.3	5.1.1.4 5.3.4	6.22 Software requirements description
SRS	Software Requirements Spec (SRS)	81433	F.2.4	5.1.1.4 5.3.4	6.22 Software requirements description, 6.27 Test or validation

				5.3.5.5 5.3.6 5.3.7 6.5	plan
SSDD	System/Subsys. Design Description (SSDD)	81432	G.2.1	5.3.3.1 5.3.3.2	6.25 Software arch.& reqts alloc descr
IDD	Interface Design Description (IDD)	81436	G.2.2	5.3.5.2 5.3.6.2	6.19 Software interface design descr
--	Database Design Description (DBDD)	81437	G.2.3	5.3.5.3 5.3.6.3 5.3.7.1	6.4 Database design description
SDD	Software Design Description (SDD)	81435	G.2.4	5.3.5 5.3.6	6.12 Software arch. description, 6.16 Software design description
STD	Software Test Description (STD)	81439	H.2.1	5.1.5.1 5.3.7.1 5.3.8 5.3.10 6.5	6.28 Test or validation procedures
STR	Software Test Report (STR)	81440	H.2.2	5.3.7.2 5.3.8.2 5.3.9.1 5.3.10.1 5.3.11.1 5.3.13.1 6.5	6.29 Test or validation results report
SPS	Software Product Specification (SPS)	81441	I.2.1	5.3.1.2 6.2.2.1	Software product description
VDD	Software Version Description (SVD)	81442	I.2.	6.2	6.13 Software config. index record
SPM	Computer Prog'mg Manual (CPM)	81447	I.2.3	--	--
FSM	Firmware Support Manual (FSM)	81448	I.2.4	--	--
SUM	Software User Manual (SUM)	81443	J.2.1	5.3.4.1 5.3.5.4 5.3.6.4 5.3.7.3 5.3.8.3 5.3.8.5 5.3.9.2	6.30 User documentation description
--	Software Input/Output Manual (SIOM)	81455	J.2.2	--	--
--	Software Center Operator Mnl (SCOM)	81444	J.2.3	5.4	6.9 Operation process plan
CSOM	Computer Operation Manual (COM)	81446	J.2.4	5.4	6.9 Operation process plan

APPENDIX D. Earned Value Management Overview

Defense Systems Management College
Earned Value Management Gold Card



VARIANCES *(Favorable is positive, Unfavorable is negative)*

- Cost Variance — CV = BCWP – ACWP — CV % = CV / BCWP
- Schedule Variance — SV = BCWP – BCWS — SV % = SV / BCWS
- Variance at Completion — VAC = BAC – EAC

PERFORMANCE INDICES *(Favorable is > 1.0, Unfavorable is < 1.0)*

- Cost Efficiency — $CPI = \frac{BCWP}{ACWP}$
- Schedule Efficiency — $SPI = \frac{BCWP}{BCWS}$

OVERALL STATUS

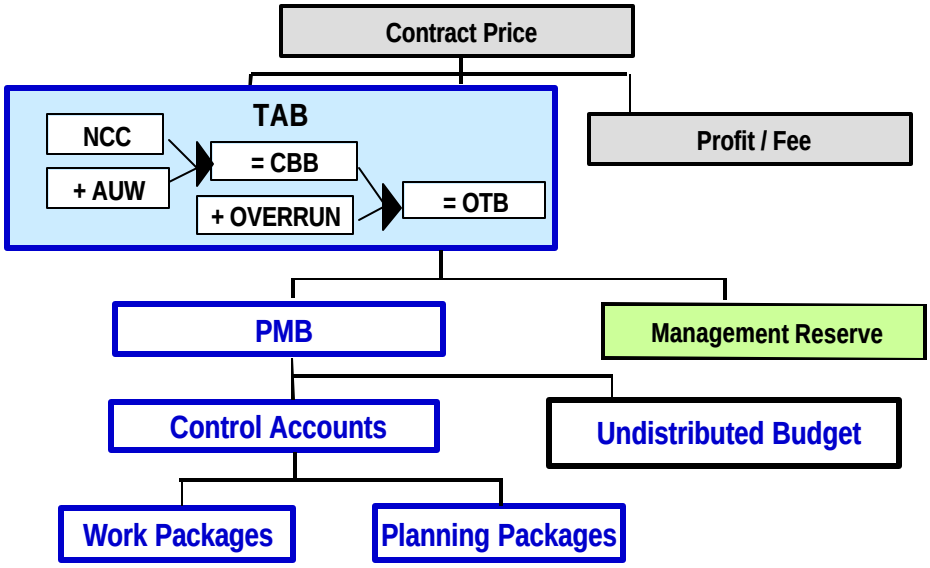
- Percent Complete = $\frac{BCWP\ CUM}{BAC}$
- Percent Spent = $\frac{ACWP\ CUM}{BAC}$

TO COMPLETE PERFORMANCE INDEX (TCPI)

$$TCPI_{(EAC)} = \frac{WORK\ REMAINING}{COST\ REMAINING} = \frac{BAC - BCWP\ CUM}{EAC - ACWP\ CUM}$$

ESTIMATE AT COMPLETION (EAC = ACWP + Estimate for Remaining Work)

$$EAC_{CPI} = \frac{BAC}{CPI\ CUM} \qquad \bullet\ EAC_{Composite} = ACWP\ CUM + \frac{BAC - BCWP\ CUM}{(CPI\ CUM) \bullet (SPI\ CUM)}$$



TERMINOLOGY

NCC	– Negotiated Contract Cost	Contract price less profit / fee
AUW	– Authorized Unpriced Work	Work authorized to start, not yet negotiated
CBB	– Contract Budget Base	Sum of NCC and AUW
OTB	– Over Target Baseline	Sum of CBB and recognized overrun
TAB	– Total Allocated Budget	Sum of all contract budgets - NCC,CBB or 0TB (includes MR)
BAC	– Budget At Completion	Cumulative BCWS - total end point of PMB (excludes MR)
PMB	– Performance Measurement Baseline	Contract time-phased, budgeted work plan (excludes MR)
MR	– Management Reserve	Contractor PM's Contingency budget
UB	– Undistributed Budget	Broadly defined activities not yet distributed to CAs
CA	– Control Account	Contractor key management control point - CWBS element
WP	– Work Package	Near-term, detail-planned activities within a CA
PP	– Planning Package	Far-term CA activities not yet defined into detail Work Packages
BCWS	– Budgeted Cost for Work Scheduled	Value of work scheduled -- PLAN
BCWP	– Budgeted Cost for Work Performed	Value of work completed -- EARNED VALUE
ACWP	– Actual Cost of Work Performed	Cost of work completed -- ACTUAL COSTS INCURRED
EAC	– Estimate At Completion	Estimate of total contract costs

EVM POLICY (DOD 5000.2-R)

ALTERNATIVE EV MANAGEMENT APPLICATIONS

LEVEL 1. EVMS Industry Standards Management Application

Contractor management system certified as meeting Industry Standards

- Required for non-FFP contract exceeding $73M RDT&E or $315M in procurement (CY00$).
- PM may apply to contracts below-threshold —consider benefits, risk and criticality.
- Contractor must establish, maintain, and use a system that meets the the 32 Industry Standards.
- Cost Performance Report (CPR) delivered as a CDRL item.
  - 5 Formats (WBS, Organization, Baseline, Staffing, and Explanations)

LEVEL 2. C/SSR Management Application

Contractor Management system not certified

- Required for non-FFP contract exceeding $6.3M (CY00$) and 12 months in length.
- 'Reasonably objective' EV methods acceptable, traceability at higher level (CA vs WP)
- The CPR or the Cost/Schedule Status Report (C/SSR) delivered as a CDRL item.

EVM Home Page — http://www.acq.osd.mil/pm/
DSMC EV E-Mail Address — EVM@DSMC.DSM.MIL
DSMC EV Phone No. — (703) 805–2848/2968 (DSN 655)

June 2000